IAS/PCMI Summer Session 2000
Clay Mathematics Undergraduate Program
Basic Course on Computational Complexity

Lecture 6: Reductions and Completeness

David Mix Barrington and Alexis Maciel
July 24, 2000

1. Definitions

As mentioned in the first week, whether P equals NP is a question of great importance. Since we cannot currently identify a single problem in NP that is not in P, can we at least identify the "hardest" problems in NP? What would that mean? If such problems exist, they should have the property that if they belong to P, then all of NP should belong to P. Now how could that be realized? One way would be for a polynomial-time algorithm for any of these "hardest" problems to give rise to polynomial-time algorithms for any of the problems in NP. Such a mechanism is formalized in the notion of a polynomial-time reduction.

A language A is *polynomial-time reducible* to a language B, written $A \leq_{\mathrm{P}} B$, if there is a polynomial-time computable function f such that for every x, $x \in A$ if and only if $f(x) \in B$. Informally, f gives us an efficient way of transforming instances of problem A into instances of problem B in such a way that we can solve one by solving the other. The function f is called a *reduction* from A to B.

The idea of a reduction is actually a common and central concept in computer science. For example, whenever we design a modular program for a certain problem, we are reducing that problem to a collection of other problems.

Note that if $A \leq_{\mathrm{P}} B$ and $B \in \mathrm{P}$, then $A \in \mathrm{P}$. Therefore, the fact that A reduces to B implies that A is no harder than B, or equivalently that B is no easier than A, in the sense that if B was easy (i.e., in P), then A would be easy too. This justifies the notation $A \leq_{\mathrm{P}} B$. Note also that polynomial-time reducibility is a transitive relation: if $A \leq_{\mathrm{P}} B$ and $B \leq_{\mathrm{P}} C$, then $A \leq_{\mathrm{P}} C$.

We now say that a language is NP-*complete* if it is in NP and every other language in NP reduces to it. If any NP-complete problem is in P, then all NP is in P. We will

therefore consider NP-complete problems as our formal notion of hardest problems in NP.

NP-completeness is an interesting concept for both theoretical and practical reasons. On the theoretical side, showing that a single NP-complete problem is in P shows that P = NP. It can also be argued that since NP-complete problems are the hardest problems in NP, then they are good candidates for trying to prove that P $\subset$ NP.

On the practical side, finding a polynomial-time algorithm for an NP-complete problem would solve an open problem in which many researchers have already invested a lot of time and energy. So discovering that a particular problem is NP-complete is probably a good indication that trying to find a polynomial-time algorithm will not be very productive. It is probably best to try to find a way around the problem or simply settle for an exponential-time algorithm.

2. An NP-Complete Problem

Finding our first NP-complete problem will not be very difficult. However, this problem will be somewhat artificial. In the next lecture, we will exhibit more natural NP-complete problems.

Given a string x of length n, a nondeterministic machine N and a string of t 1's, does N accept x in time t? We call this the NP *evaluation* problem.

Theorem 1 NP *evaluation is* NP*-complete.*

Proof First, we must show that NP evaluation is in NP. Given x, N and 1^t, we cannot blindly simulate N on x because we have no guarantee that N runs in polynomial time. So we will simulate N on x while keeping a count of the number of steps performed. As soon as that count exceeds t, the machine stops and rejects. It is clear that N accepts x in time t following a certain sequence of choices if and only if our simulator accepts following that same sequence of choices. In addition, the simulator runs in linear time.

Now we must show that every language A in NP reduces to NP evaluation. Let N be a polynomial-time machine for A. Suppose N runs in time n^k. Then $x \in A$ if and only if N accepts x in time n^k. Therefore, the reduction simply has to put together x, a description of N, and the string 1^{n^k}. This can be easily done in polynomial time. □

3. A Reduction Between Two "Natural" Problems

In the next lecture, we will prove the NP-completeness of more natural problems, not ones so obviously tied to NP. In fact, such NP-complete problems abound and, in the great majority of cases, their NP-completeness is established by a reduction from a known NP-complete problem. This relies on the fact that if A is NP-complete, $A \leq_{\mathrm{P}} B$ and $B \in \mathrm{NP}$, then B is NP-complete. Note that all NP-complete problems are *equivalent* to each other, with respect to polynomial-time reductions, in the sense that they reduce to each other.

In the meantime, as a warm-up for all this, we end this lecture with an example of a reduction between two of these natural problems. A *Boolean formula* is an expression involving Boolean variables and Boolean operations. The Boolean variables can take as values TRUE or FALSE (or 0 or 1). For the moment, we will consider the Boolean operations AND, OR and NOT, symbolically written as $\wedge$, $\vee$ and $\neg$. $\neg x$ is also often written as $\overline{x}$. For example, $(x_1 \vee x_2) \wedge (\overline{x_1} \vee x_3)$ is a formula.

A Boolean formula is *satisfiable* if it evaluates to TRUE for some truth assignment of the variables. Given a formula, the *satisfiability* problem (SAT) is to determine whether the formula is satisfiable.

3SAT is a special case of SAT in which we want to accept only formulas of a certain type. A *CNF formula* is a conjunction of disjunctions of literals, where a literal is either a variable or its negation. The above formula is an example of a CNF formula. The disjunctions are called clauses. A *3CNF formula* is one in which all clauses have exactly three literals. Given a 3CNF formula, 3SAT is the problem of determining whether the formula is satisfiable.

As defined in an earlier exercise, in an undirected graph, a set of nodes is called a *clique* if every two of its nodes is connected by an edge. Given an undirected graph G and a number k, the CLIQUE problem is to determine whether G contains a clique of size k.

Even though these problems appear to be completely unrelated, we show here that 3SAT reduces to CLIQUE.

Theorem 2 *3SAT reduces to CLIQUE.*

Proof We need to transform a 3CNF formula into a graph. There will be a node in the graph for each literal in the formula. There will be edges between every pair of nodes except (1) there are no edges between nodes that are associated with the same clause, and (2) there are no edges between nodes that correspond to contradictory

literals such as x_1 and $\overline{x_1}$. It is clear that this graph can be constructed in polynomial time.

Let k be the number of clauses in the formula. We claim that the formula is satisfiable if and only if the graph has a clique of size k. To prove this claim, first suppose that the formula is satisfiable. Choose one true literal for each clause and consider the set of corresponding nodes. That set has size k. Any two nodes in this set cannot be associated with the same clause. And they cannot correspond to contradictory literals since they both correspond to literals that evaluate to TRUE. Therefore, there must be an edge between any two nodes in the set and we have a clique of size k.

Now suppose that the graph has a clique of size k. All the nodes in this clique must be associated with different clauses. Choose an assignment to the variables that makes the corresponding literals true. This is possible since there can be no nodes in the clique that correspond to contradictory literals. We now have an assignment that satisfies makes one literal true in each clause so the formula is satisfied. □

4. Exercises

1. Show that $A \leq_{\mathrm{P}} B$ if and only if $\overline{A} \leq_{\mathrm{P}} \overline{B}$.

2. Which languages would be NP-complete if NP was equal to P? (Hint: "All of P" is not the right answer.)

3. A Boolean formula is a *tautology* if it evaluates to 1 for *all* truth assignments. Let TAUT be the set of all tautologies. Show that SAT reduces to $\overline{\mathrm{TAUT}}$, the complement of TAUT.

4. Given a binary string, the MAJORITY problem is to determine whether that string contains at least as many 1's as 0's. Show that MAJORITY reduces to the following version of the graph reachability problem: given a directed graph G, two nodes s and t and a number k, determine whether there is a path of length at most k from s to t in G.

5. Show that graph reachability reduces to SAT.

6. Show that SAT is equivalent to 3SAT. (Hint: To show that SAT reduces to 3SAT, rewrite the formula using binary operations only. Then, viewing the formula as a circuit, write a 3CNF formula that essentially expresses the fact that the values on all the wires are consistent with the gates.)

7. Show that CLIQUE reduces to SAT. (Hint: For any k, there are logspace-uniform NC^1 circuits that can take n Boolean variables and determine if at least k of them are 1.)