COMP 314 Class 24: NP-complete problems

1 CircuitSat is a "hardest" NP problem

Last time we saw several equivalent definitions for NP-completeness. The easiest definition is to start with a laundry list of famous poly-equivalent problems (e.g. HamiltonianCircuit, CircuitSat, TravelingSalesperson, SubsetSum), and *define* them to be NP-complete. Then declare that any other problem that is poly-equivalent to a previously-known NP-complete problem is also NP-complete.

If we adopt this easy definition, we miss out on one of the important properties of NP-completeness: the NP-complete problems are the "hardest" problems in NP. That is, *any* NP problem can be poly-reduced to *any* NP-complete problem. We are now going to sketch a proof of this amazing fact, by reducing an arbitrary NP problem to CircuitSat.

Claim: Let D be a problem in NP. Then there is a polytime reduction from D to CircuitSat.

Proof of the claim: D is in NP, so we can verify its solutions in polynomial time. Specifically, there is a program `verifyD.py` that accepts, as input, the concatenation of an input string I, and hint H. The length of H and the running time of `verifyD.py` are bounded by polynomials (as a function of n, the length of I). Denote these polynomials by $b(n)$ and $c(n)$, respectively. In this course, we usually work with ASCII inputs and outputs, but in this proof it will be easier to work with binary. So I and H are binary strings, and the output is a single bit (1 for "yes, I has been verified as a positive instance of D", 0 for "no, we failed to verify that I is a positive instance of D").

It's worth clarifying something before moving on. Usually, for NPoly/PolyCheck problems, the verification program would receive the concatenation of *three* strings: input string I, solution S, and hint H. But in this proof, the solution S does not appear. Why? Because D is a decision problem, so its solution is always "`yes`" or "`no`". And because of the "no" caveat in the definition of PolyCheck, we only required to efficiently verify *positive* instances. Therefore, we can *assume* S is "`yes`", so it doesn't have to appear as one of the inputs to `verifyD.py`.

Let's get back to the main proof. We are trying to reduce D to CircuitSat. As a first step, we will convert `verifyD.py` into a circuit C. Programs run on computers, and computers are made up of circuits, so this seems quite reasonable (although there are many details to be worked out). The circuit will have n inputs for I, $b(n)$ inputs for H, and a single binary output. Let's label the inputs $i_1, i_2, \ldots$ for the I-values and $h_1, h_2, \ldots$ for the H-values.

Now we have a circuit C is that mimics `verifyD.py`: by feeding the bits from I and H into the top of the circuit, we obtain an output bit at the bottom equal to the output of `verifyD.py`.

Again, recall our main objective: reduce D to CircuitSat. Given an instance of D (that is, an input I), we need to convert I to an instance of CircuitSat. Here's how we do it. Given I, first construct the circuit C described above. Then modify C to create a new circuit C', by fixing the values of $i_1, i_2, \ldots$ to the specific values of the binary string I. The new circuit has $b(n)$ inputs labeled $h_1, h_2, \ldots$. Now we ask the following question about C': "Are there any possible values of the inputs $h_1, h_2, \ldots$ that produce an output of 1?" This question is an instance of CircuitSat! And the answer is "yes" if and only if there is some hint H that causes `verifyD.py` to output a 1. This result maps perfectly onto our original problem D:

Input I is a positive instance of D

if and only if

There exists some hint H for which `verifyD.py` outputs 1

if and only if

C' is a positive instance of CircuitSat

Thus, we certainly have a reduction from D to CircuitSat. But the proof is incomplete: we haven't yet shown that the reduction can be performed in polynomial time. We won't give a rigorous proof of this, but the key point is that the number of gates in the circuit C needs to be bounded by a polynomial, as a function of n. If that's true, we can construct the circuit C, and thus C', in polynomial time. So the description of C'—which is our instance of CircuitSat—will have polynomial size.

Well, how large could C actually be? One way to convince yourself that the size remains reasonable is to think about converting each of the elementary constructs of a programming language into a small piece of circuitry. Most statements (for example, `if...else` statements) require only a fixed number of gates. The main exception to this is loops (such as `while` loops). Fortunately, the $c(n)$ bound on `verifyD.py`'s running time comes to the rescue in this case. The circuitry for the body of the loop can be repeated as many times as necessary, cutting off after $c(n)$ repeats. (This technique, which is also used by compilers, is called *unrolling* the loop.) Thus, although the details are omitted here, it can be shown that C does indeed have polynomial size, and the polytime reduction is therefore complete. ■

So, this claim tells us that for any NP problem D, we have $D \leqslant_{\text{Poly}}$ CircuitSat. In other words, when we ignore polynomial factors, CircuitSat is at least as hard as any other NP problem. And because—by our "easy" definition of NP-completeness—all the NP-complete problems are poly-equivalent to CircuitSat, this means that *any* NP-complete problem is a "hardest" NP problem.

2 Some more NP-complete problems

To give us a better taste of the phenomenon of NP-completeness, below is a list of a few more NP-complete problems, with very brief descriptions. In many cases, the problem is described as an optimization problem (e.g. find the *biggest* clique, or *minimize* the distance), because the problem is most naturally stated that way. Obviously, these optimization problems are not NP-complete—they're not even in NP, because they're not decision problems. Instead, the optimization problems listed are NP-hard, and in each case there's a closely-related variant that *is* a decision problem and *is* NP-complete.

- Clique: Find the biggest clique in a graph. (A *clique* is a set of vertices that are all connected to each other.)
- VertexCover: Find the smallest set of vertices in a graph that "covers" all the edges—that is, each edge is connected to one of the chosen vertices.
- MaxCut: Divide the nodes of a weighted graph into two subsets so that the total weight of the edges from one subset to the other is as big as possible.
- LongestPath: Find the longest (non-repeating) path in a weighted graph.
- IntegerProgramming: Find an optimal solution to a set of linear equations and inequalities, where all the variables are constrained to be integers.
- TaskAssignment: Given a set of jobs that take different amounts of time, and some *workers* (say, people or computers), assign each job to a worker so all jobs are completed as soon as possible.

3 Problems in NP but probably not NP-complete

Sometimes the rhetoric about NP-completeness starts to sound a little overwhelming. Is it really true that *all* the hard, important problems are NP-complete? The answer is widely believed (but not proven) to be "no". Here are some problems in NP that have important applications, have been studied intensively, appear to require exponential time, but have never been proven NP-complete:

- Factor: Compute the prime factorization of an integer.
- DiscreteLog: Compute $\log_b N$ using clock arithmetic.
- GraphIsomorphism: Given two graphs, determine whether they are *isomorphic* (i.e. are they actually the same graph, but with the vertices permuted?).

It's worth noting that problems related to factoring and discrete logarithm lie at the heart of some widely-used encryption systems. Graph isomorphism also has numerous applications, including compiler optimizations and certain kinds of search functionality.

4 Some related problems that are in P

And while we are in the business of deflating some of the rhetoric surrounding NP-completeness, let's remind ourselves that there are some quite remarkable algorithms solving important, difficult problems in polynomial time. A short list would include:

- PRIMALITY: Given an integer, determine whether or not it is prime.
- MINCUT: Given a weighted graph, split the vertices into two subsets so that the total weight of the edges between the subsets is minimized.
- SHORTESTPATH: Find the shortest path between vertices in a weighted graph.
- LINEARPROGRAMMING: Find an optimal solution to a set of linear equations and inequalities.

It's interesting to note that in every one of these cases, a seemingly small change to the problem catapults it from P to NP or NPComplete. Specifically:

- PRIMALITY (P) becomes FACTOR (NP, believed to require exponential time) if we have to *find* a factor rather than merely detecting its existence.
- MINCUT (P) becomes MAXCUT (NPComplete) if we look for the largest, rather than smallest cut.
- SHORTESTPATH (P) becomes LONGESTPATH (NPComplete) if we look for the longest, rather than shortest path.
- LINEARPROGRAMMING (P) becomes INTEGERPROGRAMMING (NPComplete) if we insist on integer-valued, rather than real-valued variables.

5 Some NP-hard problems can be approximated efficiently

Here's some more good news: despite the fact that we can't solve them exactly, some NP-hard problems can be *approximated* efficiently. For example, there is a polynomial time algorithm guaranteed to find a solution to TRAVELINGSALESPERSON at most 50% longer than the optimal solution. And the TASKASSIGNMENT problem defined above, despite being NP-hard, can be approximated arbitrarily well! That is, giving any desired accuracy—say, 1%—there's a polytime algorithm that finds a solution of the desired accuracy.

6 Some NP-hard problems can be solved efficiently for real-world inputs

Our last piece of good news to combat the gloom of NP-completeness is that there are some NP-hard problems that have efficient algorithms *in practice*. This means that the worst-case running time of the algorithm is exponential, but the algorithm happens to run efficiently on large instances of the problem that are actually encountered in the real world. The most famous example of this is SAT. Solving SAT problems is a huge industry in its own right. There are annual contests held for various types of real-world SAT problems, and researchers are constantly coming up with improved SAT-solvers that perform well in these contests on problem instances involving millions of clauses.