Lecture 22:
The NP-Completeness Challenge

Steven Skiena

Department of Computer Science
State University of New York
Stony Brook, NY 11794–4400

http://www.cs.sunysb.edu/∼skiena

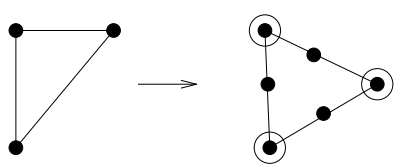
Problem of the Day

Show that the *Hitting Set* problem is NP-complete:
Input: A collection C of subsets of a set S, positive integer k.
Question: Does S contain a subset S' such that $|S'| \leq k$ and each subset in C contains at least one element from S'?

Techniques for Proving NP-completeness

1. *Restriction* – Show that a special case of your problem is NP-complete. E.g. the problem of finding a path of length k is really Hamiltonian Path.

2. *Local Replacement* – Make local changes to the structure. An example is the reduction $SAT \propto 3 - SAT$. Another is showing isomorphism is no easier for bipartite graphs:



   For any graph, replacing an edge with makes it bipartite.

3. *Component Design* - These are the ugly, elaborate constructions

The Art of Proving Hardness

Proving that problems are hard is an skill. Once you get the hang of it, it is surprisingly straightforward and pleasurable to do. Indeed, the dirty little secret of NP-completeness proofs is that they are usually easier to recreate than explain, in the same way that it is usually easier to rewrite old code than the try to understand it.

I offer the following advice to those needing to prove the hardness of a given problem:

Make your source problem as simple (i.e. restricted) as possible

Never use the general traveling salesman problem (TSP) as a target problem. Instead, use TSP on instances restricted to the triangle inequality. Better, use Hamiltonian cycle, i.e. where all the weights are 1 or ∞. Even better, use Hamiltonian path instead of cycle. Best of all, use Hamiltonian path on directed, planar graphs where each vertex has total degree 3. All of these problems are equally hard, and the more you can restrict the problem you are reducing, the less work your reduction has to do.

Make your target problem as hard as possible

Don’t be afraid to add extra constraints or weights or freedoms in order to make your problem more general (at least temporarily).

Select the right source problem for the right reason

Selecting the right source problem makes a big difference is how difficult it is to prove a problem hard. This is the first and easiest place to go wrong.

I usually consider four and only four problems as candidates for my hard source problem. Limiting them to four means that I know a lot about these problems – which variants of these problems are hard and which are soft. My favorites are:

- 3-Sat – that old reliable. . . When none of the three problems below seem appropriate, I go back to the source.
- Integer partition – the one and only choice for problems whose hardness seems to require using large numbers.

- Vertex cover – for any graph problems whose hardness depends upon *selection*. Chromatic number, clique, and independent set all involve trying to select the correct subset of vertices or edges.
- Hamiltonian path – for any graph problems whose hardness depends upon *ordering*. If you are trying to route or schedule something, this is likely your lever.

Amplify the penalties for making the undesired transition

You are trying to translate one problem into another, while making them stay the same as much as possible. The easiest way to do this is to be bold with your penalties, to punish anyone trying to deviate from your proposed solution. “If you pick this, then you have to pick up this huge set which dooms you to lose.” The sharper the consequences for doing what is undesired, the easier it is to prove if and only if.

Think strategically at a high level, then build gadgets to enforce tactics.

You should be asking these kinds of questions. “How can I force that either A or B but not both are chosen?” “How can I force that A is taken before B?” “How can I clean up the things I did not select?”

Alternate between looking for an algorithm or a reduction if you get stuck

Sometimes the reason you cannot prove hardness is that there is an efficient algorithm to solve your problem! When you can't prove hardness, it likely pays to change your thinking at least for a little while to keep you honest.

Now watch me try it!

To demonstrate how one goes about proving a problem hard, I accept the challenge of showing how a proof can be built on the fly.

I need a volunteer to pick a random problem from the 400+ hard problems in the back of Garey and Johnson.

The Problem

The Solution

Other NP-complete Problems

- Partition - can you partition n integers into two subsets so that the sums of the subset are equal?
- Bin Packing - how many bins of a given size do you need to hold n items of variable size?
- Chromatic Number - how many colors do you need to color a graph?
- $N \times N$ checkers - does black have a forced win from a given position?

Open: Graph Isomorphism, Factoring Integers.

Polynomial or Exponential?

Just changing a problem a little can make the difference between it being in P or NP-complete:

	P	NP-complete
	Shortest Path	Longest Path
	Eulerian Circuit	Hamiltonian Circuit
	Edge Cover	Vertex Cover

The first thing you should do when you suspect a problem might be NP-complete is look in Garey and Johnson, *Computers and Intractability*.