6.080/6.089 GITCS March 13, 2008

Lecture 11

Lecturer: Scott Aaronson *Scribe: Michael Fitzgerald*

1 NP-Completeness In Practice

Recent lectures have talked a lot about technical issues. This lecture is going to take a step back.

We've shown that **NP**-complete problems do occur in real-life problems and situations, so obviously engineers, scientists, etc. have come across them while doing their jobs. For example, trying to set up an optimal airline flight schedule, or an optimal conflict-minimizing class schedule, or trying to design the optimal airplane wing. When you run up against an **NP**-complete problem, however, do you just give up? No, there are many ways to cope with an encounter with an **NP**-complete problem. You have to change the way you look at the problem; you stop looking for a general-purpose solution and play a different game.

1.1 Alternatives

We'll discuss seven strategies for dealing with **NP**-complete problems.

1.1.1 Brute Force

This works fine if you have a small instance of a problem; you just try every possible solution until you find one that works. Your problem could be **NP**-complete, but if your n is 4, finding the solution is trivial. For example, the sudoku in your Sunday paper is possible to brute force; you can find a solution for a nine by nine grid, but huge ones are much more difficult.

1.1.2 Structure Exploitation

Your particular instance of the **NP**-complete problem might have a structure that lends itself to a solution. Remember, "**NP**-complete" is a statement about the while *class* of problems; it doesn't say anything about the complexity of an individual instance. "Proof With n Symbols" is an **NP**-complete problem, but humans have quite obviously solved many specific instances of it.

1.1.3 Special Casing

You can find special cases of the general problem that both lend themselves to a general solution and contain your instance of the general problem. For example, $2SAT$ is a special case of SAT that is solvable; it has a special form that has a polynomial-time solution algorithm. Every **NP**-complete problem has some special cases that can be solved in polynomial-time; you just have to find a special subset to which your instance belongs.

1.1.4 Taking Advantage of Parameters Besides n

This strategy is related to the previous one. Traditionally, we consider everything to be a function of n, the size of the input. But we can also consider other parameters. As an example, recall the max-clique problem: given a set of people, find the largest subset that are all friends with each other. This is an **NP**-complete problem. But suppose we add another parameter k, and change the question to whether there's a clique of size k. If k is given as part of the input, then this problem is still **NP**-complete. (As a sanity check, if $k = n/2$, then the number of possibilities to check is something like $\binom{n}{n/2}$, which grows exponentially with n.) If k is a constant, however, the problem *does* have an efficient solution: you can simply check all sets of size k, which takes $\binom{n}{k}$ time. So, for every fixed k, there's a polynomial-time algorithm to solve the problem.

Indeed, in some cases one can actually do better, and give an algorithm that takes $f(k)p(n)$ time, for some function f and polynomial p (so that the parameter k no longer appears in the exponent of n). The study of the situations where this is and isn't possible leads to a rich theory called *parameterized complexity*.

1.1.5 Approximation

If we can't get an optimal solution, sometimes we can get one that's close to optimal. For example, in the max-clique problem, can we find a clique that's at least half the size of the biggest one? In the traveling salesman problem, can we find a route that's at most twice as long as the shortest route? The answers to these questions will in general depend on the problem.

Here's a more in-depth example. We know that $3SAT$ is **NP**-complete, but is there an efficient algorithm to satisfy (say) a $7/8$ fraction of the clauses? If we fill in each of the variables randomly, we have a $1/8$ chance that each individual clause is false. So on average we're going to satisfy about $7/8$ of the clauses, by linearity of expectation.

On the other hand, is there an efficient algorithm to satisfy a $7/8 + \epsilon$ fraction of the clauses, for some constant $\epsilon > 0$? A deep result of Håstad (building on numerous contributions of others) shows that this is already an **NP**-complete problem—that is, it's already as hard as solving $3SAT$ perfectly. This means that $7/8$ is the limit of approximability of the $3SAT$ problem, unless **P**=**NP**.

The approximability of **NP**-complete problems has been a huge research area over the last twenty years, and by now we know a quite a lot about it (both positive and negative results). Unfortunately, we won't have time to delve into the topic further in this course.

1.1.6 Backtracking

We can make the brute force method more effective by generating candidate solutions in a smarter way (in other words, being less brute). In the 3-coloring problem, for example, you might know early on that a certain solution won't work, and that will let you prune a whole portion of the tree that you might otherwise have wasted time on. In particular, given a map, you can fill in colors for each of the countries, branching whenever more than one color can work for a given country. If you reach a conflict that invalidates a potential solution, go back up the tree until you reach an unexplored branch, and continue on from there. This algorithm will eventually find a solution, provided one exists—not necessarily in polynomial time, but at least in exponential time with a smaller-order exponential.

1.1.7 Heuristics

The final strategy involves using heuristic methods such as simulated annealing, genetic algorithms, or local search. These are all algorithms that start with random candidate solutions, and then repeatedly make local changes to try and decrease "badness." For example, with 3-coloring, you might start with a random coloring and then repeatedly change the color of a conflicting country, so that the conflicts with its neighbors are fixed. Rinse and repeat.

This sort of approach will not always converge efficiently on a good solution—the chief reason being that it can stuck on local optima.

1.2 In Nature

Nature deals with **NP**-complete problems constantly; is there anything we can learn from it? Biology employs what computer scientists know as genetic algorithms. Sometimes biology simply mutates the candidate solutions, but other times it combines them (which we call sexual reproduction).

Biology has adopted the sexual reproduction technique so widely that it *must* have something going for it, even if we don't yet fully understand what it is. In experiments, local search algorithms that employ only mutations (i.e. "asexual reproduction") consistently do better at solving **NP**-complete problems than do algorithms that incorporate sex.

The best theory people currently have as to why nature relies so heavily on sex is that nature isn't trying to solve a fixed problem; the constraints are always changing. For example, the features that were ideal for an organism living in the dinosaur age are different from the features that are ideal for modern-day organisms. This is still very much an open issue.

As another example of nature apparently solving **NP**-complete problems, let's look at protein folding. Proteins are one of the basic building blocks of a cell. DNA gets converted into RNA, which gets converted into a protein, which is a chain of amino acids. We can think of that chain of amino acids as just encoded information. At some point, however, that information needs to be converted into a chemical that interacts with the body in some way. It does this by folding up into a molecule or other special shape that has the requisite chemical properties. The folding is done based on the amino acids in the chain, and it's an extremely complicated process.

The key computational problem is the following: given a chain of amino acids, can you predict how it's going to fold up? In principle, you could simulate all the low-level physics, but that's computationally prohibitive. Biologists like to simplify the problem by saying that every folded state has potential energy, and the folding is designed to minimize that energy. For idealized versions of the problem, it's possible to prove that finding a minimum-energy configuration is **NP**-complete. But is every cell in our body really solving hard instances of **NP**-complete problems millions of times per second?

There is another explanation. Evolution wouldn't select for proteins that are extremely hard to fold (e.g., such that any optimal folding must encode a proof of the Riemann Hypothesis). Instead, it would select for proteins whose folding problems are computationally easy, since those are the ones that will fold reliably into the desired configuration.

Note that when things go wrong, and the folding *does* get caught in a local optimum, you can have anomalies such as prions, which are the cause of Mad Cow Disease.

2 Computational Universe Geography

We finished previously by mentioning Ladner's Theorem: if $\mathbf{P} \neq \mathbf{NP}$, then there are **NP** problems that are in neither in **P** nor **NP**-complete. The proof of this theorem is somewhat of a groaner; the basic idea is to define a problem where for some input sizes n the problem is to solve SAT (keeping the problem from being in **P**), and for other input sizes the problem is to do nothing (keeping the problem from being **NP**-complete). The trick is to do this in a careful way that keeps the problem in **NP**.

While the problem produced by Ladner's theorem isn't one we'd ever care about in practice, there *are* some problems that are believed to be intermediate between **P** and **NP**-complete and that matter quite a bit. Perhaps the most famous of these is factoring.

2.1 Factoring

We don't yet know how to base cryptography on **NP**-complete problems. The most widely used cryptosystems today are instead based on number theory problems like factoring, which—contrary to a popular misconception—are neither known nor believed to be **NP**-complete.

Can we identify some difference between factoring and the known **NP**-complete problems, that might make us suspect that factoring is *not* **NP**-complete?

One difference involves the number of solutions. If we consider an **NP**-complete problem like 3-coloring, there might be zero ways to 3-color a given map, a few ways, or an enormous number of ways. (One thing we know is that the number of colorings has to be divisible by 6—do you see why?) With factoring, by contrast, there's exactly one solution. Every positive integer has a unique prime factorization, and once we've found it there aren't any more to find. This is very useful for cryptography, but it also makes it extraordinarily unlikely that factoring is **NP**-complete.

2.2 coNP

Why do I say that?

Well, let's forget about factoring for the moment and ask a seemingly unrelated question. Supposing that there's no efficient algorithm to find a 3-coloring of a graph, can we at least give a short proof that a graph isn't 3-colorable? If the backtracking tree is short, you could just write it down, but that would take exponential time in the worst case. The question of whether there exist short proofs of *un*satisfiability, which can be checked in polynomial time, is called the **NP** versus **coNP** question. Here **coNP** is the set of *complements* of **NP** problems: that is, the set $\left\{\overline{L} : L \in \mathbf{NP}\right\}$.

Relating this back to the original **P** versus **NP** question, if **P**=**NP** then certainly **NP**=**coNP** (if we can decide satisfiability in polynomial time, then we can also decide unsatisfiability!). We're not sure, however, if **NP**=**coNP** implies **P**=**NP**; we don't know how to go in this direction. Still, almost all computer scientists believe that **NP**≠**coNP**, just as they believe **P**≠**NP**.

2.3 New Universe Map

We now have five main regions on our universe map: **P**, **NP**, **NP**-complete, **coNP**, and **coNP**-complete.

If a problem is in both **NP** and **coNP**, does that mean it's in **P**? Well, consider a problem derived from factoring, like the following: given a positive integer n, does n have a prime factor that ends in 7? This problem is in both **NP** and **coNP**, but we certainly don't know it to be in **P**.

Finally, here's the argument for factoring not being **NP**-complete. Because of the existence of unique factorization, factoring is in **NP**∩**coNP**. So if factoring were **NP**-complete, then an **NP**-complete problem would be in **coNP**. So **NP** itself would be contained in **coNP**, and (by symmetry) would equal **coNP**—thereby making a large part of the computational universe collapse. To put it differently, if **NP**≠**coNP**, then factoring can't be **NP**-complete.