MIT OpenCourseWare
http://ocw.mit.edu

6.080 / 6.089 Great Ideas in Theoretical Computer Science
Spring 2008

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.

6.080/6.089 GITCS	Mar 11, 2008

Lecture 10

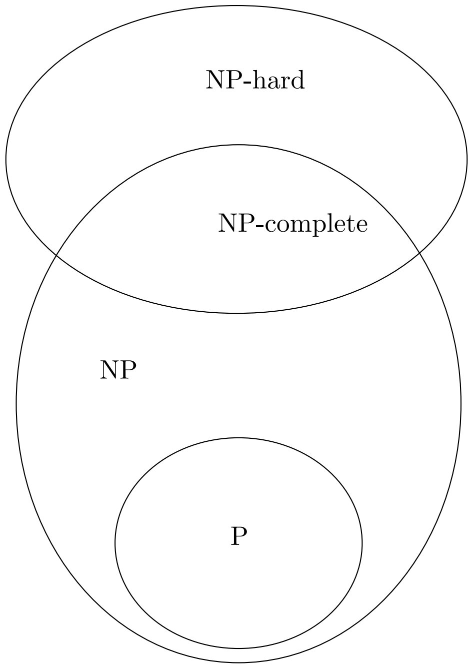
Lecturer: Scott Aaronson — *Scribe: Yinmeng Zhang*

1 Administrivia

Work hard on the homework, but don't freak out. Come to office hours. There will be a short (1 week) pset before spring break, and an exam on the Thursday after, which is April 3rd.

2 Recap

Last time we saw a rough geography of the universe of computational problems.



One of the precepts of this course is that this picture is really important. In an ideal world, people would recognize this map the same way they recognize a map of the continents of earth. As part of the popularization effort we are distributing Scott's Scientific American article on the limits of quantum computation, which contains a similar picture, except quantum-ier.

The concept of **NP**-completeness was defined in the '70s. Last time we saw that the DUH problem — given a Turing machine, is there an input on which it halts in polynomial time — is obviously **NP**-complete, duh. We then took a look at our first "natural" **NP**-complete problem, SATISFIABILITY (or SAT for short). We saw Cook and Levin's proof that SAT is **NP**-complete by reduction from DUH. Today we will see some more reductions.

3 SAT reduces to 3SAT

Though SAT is **NP**-complete, specific instances of it can be easy to solve.

Some useful terminology: a *clause* is a single disjunction; a *literal* is a variable or the negation of a variable. A Boolean formula in *conjunctive normal form* (CNF) is one which is the AND of ORs of literals. Then the 2SAT problem (which we saw earlier in the course) is defined as follows:

- Given a CNF formula where every clause involves at most 2 literals, does it have a satisfying assignment?

In Lecture 2 we saw that 2SAT is solvable in polynomial time, because we can draw the implication graph and check it for cycles in polynomial time.

Today, we'll talk about the 3SAT problem.

- Given a CNF formula where every clause involves at most 3 literals, does it have a satisfying assignment?

Unlike 2SAT, the 3SAT problem appears to be hard. In fact, 3SAT is **NP**-complete — this special case encapsulates all the difficulty of generic SAT!

To prove this, we'll show how to convert an arbitrary Boolean formula into a 3SAT problem in polynomial time. Thus, if we had an oracle for 3SAT, then we could solve SAT by converting the input into 3SAT form and querying the oracle.

So how do we convert an arbitrary Boolean formula into a 3SAT formula? Our first step will be to convert the formula into a circuit. This is actually really straightforward – every $\neg$, $\wedge$, or $\vee$ in the formula becomes a NOT, AND, or OR gate in the circuit. One detail we need to take care of is that when we say multiple literals $\wedge$ed or $\vee$ed together, we first need to specify an order in which to take the $\wedge$s or $\vee$s. For example, if we saw $(x_1 \vee x_2 \vee x_3)$ in our formula, we should parse it as either $((x_1 \vee x_2) \vee x_3)$ which becomes $OR(x_1, OR(x_2, x_3))$, or $(x_1 \vee (x_2 \vee x_3))$ which becomes $OR(OR(x_1, x_2), x_3)$. It doesn't matter which one we pick, because $\wedge$ and $\vee$ are both associative.

So every Boolean formula can be converted to an equivalent circuit in linear time. This has a couple of interesting consequences. First, it means the problem CircuitSAT, where we are given a circuit and asked if it has a satisfying assignment, is **NP**-complete. Second, because Cook-Levin gave us a way to convert Turing Machines into Boolean formulas if we knew a bound on the run time, tacking this result on gives us a way to convert Turing Machines into circuits. If the Turing Machine ran in polynomial time, then the conversion will only polynomial time, and the circuit will be of polynomial size.

Ok, so now we have a circuit and want to convert it into a 3SAT formula. Where does the 3 come in? Each gate has at most 3 wires attached to it – two inputs and one output for AND and OR, and one input and one output for NOT. Let's define a variable for every gate. We want to know if there is a setting of these variables such that for every gate, the output has the right relationship to the input/inputs, and the final output is set to true.

So let's look at the NOT gate. Call the input x and the output y. We want that if x is true then y is false, and if x is false then y is true – but we've seen how to write if-then statements as disjunctions! So, we have

$$NOT : (x \vee y) \wedge (\neg x \vee \neg y)$$

We can do the same thing with the truth table for OR. Call the inputs x_1 and x_2 and the output y. Let's do one row. If x_1 is true and x_2 is false, then y is true. We can write this as

$(\neg(x_1 \wedge \neg x_2) \vee y)$. By DeMorgan's Law, this is the same as $(\neg x_1 \vee x_2 \vee y)$. Taking the AND over all possibilities for the inputs we get

$$OR : (\neg x_1 \vee \neg x_2 \vee y) \wedge (\neg x_1 \vee x_2 \vee y) \wedge (x_1 \vee \neg x_2 \vee y) \wedge (x_1 \vee x_2 \vee \neg y)$$

And similarly for the AND gate. The big idea here is that we're taking small pieces of a problem we know to be hard (gates in a circuit) and translating them into small pieces of a problem we're trying to show is hard (CNF formulas). These small pieces are called "gadgets", and they're a recurring theme in reduction proofs. Once we've got these gadgets, we have to somehow stick them together.

In this case, all we have to do is AND together the Boolean formulas for each of the gates, and the variable for the final output wire. The Boolean formula we get from doing this is true if and only if the circuit is satisfiable. Woo.

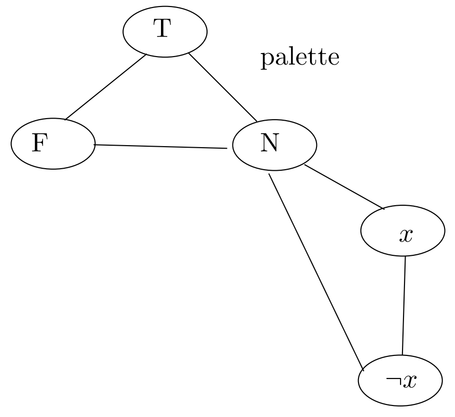
4 3COLOR is NP-complete

- Given a graph, is there a way to color each vertex either Red, Blue, or Green, so that no adjacent vertices end up the same color?

This problem is called 3COLOR, and it is **NP**-complete. We'll prove it by showing a reduction from CircuitSAT. So suppose we're given a circuit, and we have a magic box that, when given a graph, says whether it has a good 3-coloring. Can we make a graph that is 3-colorable iff the Boolean formula is satisfiable? We're going to need some gadgets.

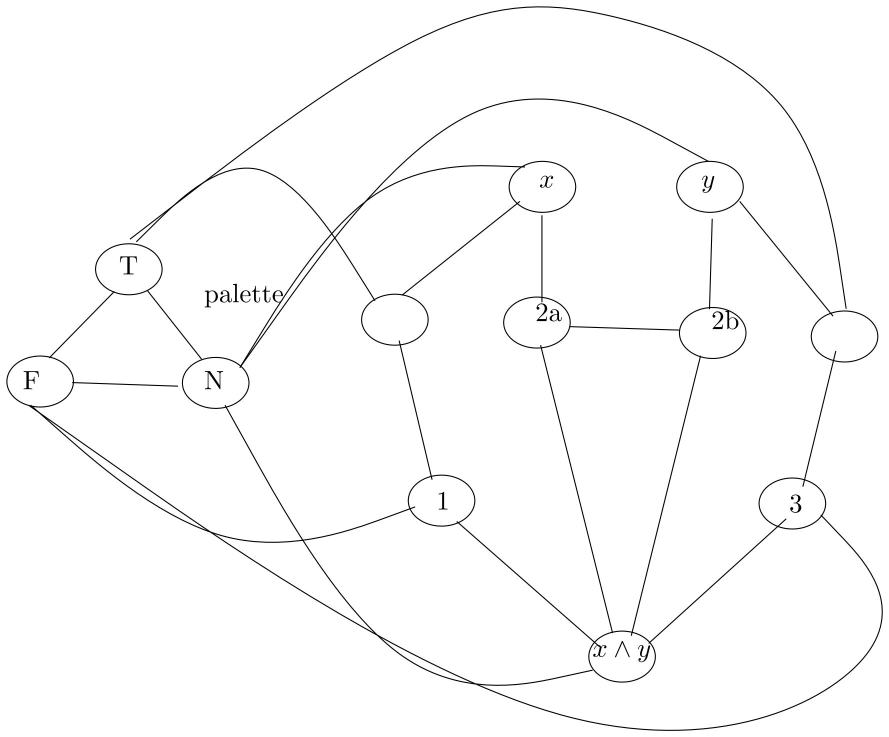
First of all, we're renaming the colors. The first piece of our graph is going to be a triangle, which will serve as a palette. Notice that any good 3-coloring must assign different colors to each vertex of a triangle. So for a given good 3-coloring, whatever color the first vertex gets assigned, we're going to call True; the second is going to be False; and the third is going to be Neither.

Here is a gadget for the NOT gate.



Because x and $\neg x$ are connected to each other, any good coloring will assign them different colors, and because they're both connected to the Neither vertex in the palette, one of them will have to be true, and one of them will be false.

Here is a gadget for the AND gate, courtesy of Alex.



It's obvious it works, right? The main things to note are as follows. The inputs and output are connected to Neither, which forces them to take on boolean values. If x and y are colored the same, then $2a$ and $2b$ will be colored with the other two colors, forcing $x \wedge y$ will have to be colored the same as x and y – that's correct behavior for an AND! If x and y are colored oppositely, then either 1 or 3 will have to be colored True, which forces $x \wedge y$ to be colored False – that's again what we were hoping would happen. Finally, in all cases, we haven't added too many restrictions: good colorings exist for any setting of x and y.

A gadget for the OR gate could be made by composing the AND and NOT gadgets using DeMorgan's Law. There's also a cute way to modify the AND gadget to turn it into the OR gadget. Do you see it?

By composing these gadgets, we can turn any circuit into a graph so that assignment of T/Fs to the wires translates to colorings of the graph. If we then connect the final output vertex to the False vertex in the palette, then the graph we've constructed will have a good 3-coloring iff the circuit had a satisfying assignment.

So we've just proved 3COLOR is **NP**-complete. This is a strange thing. It means, for example, that we can efficiently construct graphs which are 3-colorable iff there's a short proof of the Riemann Hypothesis in ZF set theory. Chew on that.

Looking at the somewhat horrifying AND gadget, you might wonder if there's a way to draw the graph so that no lines cross. This turns out to be possible, and implies that 3PLANAR-COLOR (given a planer graph, is it 3-colorable) is also **NP**-complete. What about 4PLANAR-COLOR? This problem turns out to be easy. In fact, you can just output yes every time! It was a long-standing conjecture that every planar graph could be colored with 4 colors. It was finally proven by Appel and Haken in 1976. There is some drama surrounding this proof that you can read about on Wikipedia. What about 2PLANAR-COLOR? This is easy too. It's a special case of 2COLOR, which we saw a greedy algorithm for last time. When it comes to planar graph-coloring, it seems there's something special about the number 3.

5 In conclusion

In 1972, Karp showed 21 problems were **NP**-complete using the above techniques and a good dose of cleverness. We could spend a month looking at clever gadgets, but we're not going to. The takeaway lesson is that **NP**-completeness is ubiquitous. Today, thousands of practical problems have been proved **NP**-complete. In fact, any time an optimization problem arises in industry, etc., a good heuristic is to assume it's **NP**-complete until proven otherwise! This, of course, is what motivates the question of whether **P**=**NP** — what makes it one of the central questions in all of math and science.

6 Tricky Questions

Cook defined a language A to **NP**-complete when it itself was in **NP**, and every **NP** problem could be solved in polynomial time if given oracle access to A.

- A is in **NP**
- SAT is in $\mathbf{P}^A$

This definition allows us to call the oracle multiple times, but in fact in the above proofs we only ever called it once. The proofs had a very specific form: somehow come up with an equivalent problem instance in the candidate language; call the oracle on it. Karp noticed this and proposed a different notion of **NP**-completeness

- There exists a polynomial time algorithm that maps satisfiable formulas to strings in A, and unsatisfiable formulas to strings outside A.

Are these definitions the same or is the second one more restrictive? This is an open problem, which you are welcome to close for extra credit.

Now, if **P**=**NP**, then every problem in **NP** will be solvable efficiently, and every problem in **NP** will be **NP**-complete: the computational universe will look pretty monotonous. If **P**≠**NP**, then what do we know? All the problems in **P** will be easy, and all **NP**-complete problems will be hard. Would there be anything in between? Would there be problems in **NP** which are neither **NP**-complete nor solvable efficiently? In 1975 Ladner showed that the answer is yes. Tune in next time to find out more.