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.

Lecture 9

Lecturer: Scott Aaronson *Scribe: Ben Howell*

1 Administrivia

1.1 Problem Set 1

Overall, people did really well. If you're not happy with how you did, or there's something you don't understand, come to office hours on Monday. No one got the challenge problem to prove the completeness of resolution, but many people got the problem with the cannonballs. The solution to the cannonball problem involves defining a function on the state of the system that is strictly monotonic. Here are two possible ways to construct that proof:

Solution 1: Let S_i be the number of cannonballs in stack i, and define a function that describes the state of the system: $f = \sum_i (S_i)^2$. Each time a stack is exploded, the function f increases by two. Since explosion is the only defined operation and it always increases the value of f, the system can never return to a previous state.

Solution 2: Let p be the rightmost stack that has been exploded so far. Then stack $p + 1$ has more cannonballs than it had at the beginning. The only way for that to go back down is to explode stack $p + 1$, which will cause stack $p + 2$ to have more cannonballs than it had at the beginning. This leads to an infinite regress, implying that there is no finite sequence of moves that returns to the starting configuration.

2 Review

The central question of theoretical computer science is: "Which problems have an efficient algorithm to solve them?" In computer science, the term "efficient" equates to "polynomial time".

Here are a few examples of problems that can be solved efficiently:

- Arithmetic, sorting, spellcheck, shortest path
- Solving linear systems (and inverting matrices)
- Linear programming
- Dynamic programming
- Stable marriage, maximum matching
- Primality testing
- Finding roots of polynomials
- 2-coloring a graph

And then there are other problems that have no known polynomial-time algorithm, for example:

- 3-Coloring a map
- Max Clique: Given a set of people, find the largest set who are all friends with each other.
- Packing: Given boxes of various dimensions, find the largest set you can pack in your truck.
- Traveling Salesman: Given a set of cities and costs of travel between cities, find the least expensive route that goes through every city.

3 General Problems

The problems listed above seem like puzzles. Let's consider some other problems that seem "deeper" or more "general":

- **The "DUH" Problem:** Given n, k, and a Turing machine M, is there an input y of size n such that $M(y)$ accepts in fewer than n^k steps?
- **The Theorem Problem:** Given a mathematical statement and an integer n, is there a proof in ZF set theory with at most n symbols?

All of these problems have a few things in common:

- They are all computable, but the computation might take an exponential amount of time.
- Given a proposed solution, it is possible to check whether that solution is correct in a polynomial amount of time.

3.1 The Theorem Problem

Let's take a closer look at the Theorem problem. Without the "at most n symbols" constraint, would this problem even be computable?

Claim: If you could solve this problem without that constraint, then you could solve the halting problem.

Proof: Given a Turing machine M, use your algorithm to find the answers to the following questions:

- Is there a proof that M halts? If so, then M halts.
- Is there a proof that M does not halt? If so, then M does not halt.

If neither proof exists, then we can still conclude that M does not halt. If it did halt, then there must be a proof that it does; the most straightforward one being to simulate M until it halts.

Therefore, we can conclude that without the "at most n symbols" constraint, this problem is not computable.

However, this problem computable with the "at most n symbols" constraint. In the worst case, the algorithm could simply search through all possible proofs with at most n symbols.

Is there a better way to solve the problem that doesn't involve a brute-force search? Exactly this question was asked in one of the most remarkable documents in the history of theoretical computer science: a letter that Kurt Gödel sent to John von Neumann in 1956. (The full text of this letter is in the appendix.) In this letter, Gödel concludes that if there were some general way to avoid brute-force search, mathematicians would be out of a job! You could just ask your computer whether there's a proof of Goldbach, the Riemann Hypothesis, etc. with at most a billion symbols.

4 P and NP

After 52 years, we still don't know the answer to Gödel's question. But there's something remarkable that we *do* know. Look again at all of these problems. A priori, one might think brute-force search is avoidable for problem X, not for problem Y, etc. But in the 1970s, people realized that in a very precise sense, *they're all the same problem.* A polynomial-time algorithm for any one of them would imply a polynomial-time algorithm for all the rest. And if you could prove that there was no polynomial-time algorithm for one of them, that would imply that there is no polynomial-time algorithm for all the rest.

That is the upshot of the **theory of NP-completeness**, which was invented by Stephen Cook and Richard Karp in the US around 1971, and independently Leonid Levin in the USSR. (During the Cold War, there were a lot of such independent discoveries.)

4.1 P and NP

P is (informally) the class of all problems that have polynomial-time algorithms. For simplicity, we focus on decision problems (those having a yes-or-no answer).

Formally: **P** is the class of all sets $L \subseteq \{0,1\}^n$ for which there exists a polynomial-time algorithm A such that $A(x) = 1$ iif $x \in L$.

NP (Nondeterministic Polynomial-Time) is (informally) the class of all problems for which a solution can be verified in polynomial time.

Formally: **NP** is the class of all sets $L \subseteq \{0,1\}^n$ for which there exists a polynomial-time Turing machine M, and polynomial p, such that $\forall x \in \{0,1\}^n : x \in L$ iff $\exists y \in \{0,1\}^{p(n)}$ s.t. $M(x,y)$ accepts (here, x is the instance of the problem and y is some solution to the problem).

Why is it called "nondeterministic polynomial-time"? It's a somewhat archaic name, but we're stuck with it now. The original idea was that you would have a polynomial-time Turing machine that can make *nondeterministic transitions* between states, and that accepts if and only if there exists a path in the computation tree that causes it to accept. (Nondeterministic Turing machines are just like the nondeterministic finite automata we saw before.) This definition is equivalent to the formal one above. First have the machine guess the solution y, and then check whether y works. Since it is a *nondeterministic* Turing machine, it can try all of the possible y's at once and find the answer in polynomial time.

Question: is $P \subseteq NP$? Yes! If someone gives you a solution to a P problem, you can ignore it and solve it yourself in NP time.

4.2 NP-hard and NP-complete

NP-hard is the class of problems that are "at least as hard as any NP problem." Formally, L is **NP-hard** if given an oracle for L, you could solve every NP problem in polynomial time. In other words, L is **NP-hard** if $NP \subseteq P^L$.

NP-complete is the class of problems that are both NP-hard and in NP. Informally, NP-complete problems are the "hardest problems in NP" — NP problems that somehow capture the difficulty of all other NP problems.

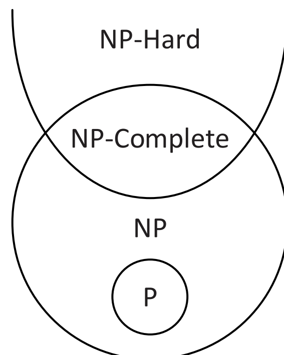

Figure 1: Relationship between P, NP, NP-Hard, and NP-Complete.

Claim: If any NP-complete problem is in P, then all are, and P = NP.
If any NP-complete problem is not in P, then none are, and P ≠ NP.

4.3 NP-complete problems

How can we prove that NP-complete problems even exists? Can we find an example of one?

Consider the "DUH" problem from above: "Given a polynomial-time Turing machine M, find an input y that causes $M(y)$ to accept." This problem is NP-complete almost by the definition of NP-complete.

- DUH is in NP: Given a solution y, it is possible to check it in polynomial time simply by running $M(y)$ to see if it accepts.
- DUH is NP-hard: Suppose we have an oracle to solve "DUH". Then given any NP problem, by the definition of NP, there must be some polynomial-time Turing machine $M\prime(x, y)$ for verifying a claimed solution y. So just ask the oracle whether there exists a y that causes M to accept (having hardwired the input x into the description of M).

So the real achievement of Cook, Karp, and Levin was not to show that NP-complete problems exist. That is almost tautological. Their real achievement was to show that many natural problems are NP-complete.

4.4 Proving a problem is NP-complete

To show a given problem L is NP-complete, you only have to do two things:

1. Show L $\in$ NP.

2. Show some problem K already known to be NP-complete reduces to L. This implies that L is NP-hard, since it must be at least as hard as K for K to reduce to L.

5 The Cook-Levin Theorem

Consider another problem: SAT (for Satisfiability). Given an arbitrary Boolean formula (in propositional logic), is there any way of setting the variables so that the formula evaluates to `TRUE`?

Theorem. SAT is a NP-complete problem.

Proof. First, show SAT is in NP. (This is almost always the easiest part.) Given a proposed set of arguments, plug them into the equation and see if it evaluates to `TRUE`.

Next, show that some *known* NP-complete problem reduces to SAT to show that SAT is NP-hard. We know DUH is NP-complete, so let's reduce DUH to SAT. In other words: given some arbitrary polynomial-time Turing machine M, we need to create a Boolean formula F, which will be satisfiable if and only if theres an input y that causes M to accept. And the translation process itself will have to run in polynomial time.

The way to do this is very simple in concept, but the execution is unbelievably complicated, so we'll focus more on the concept.

Remember that a Turing machine consists of this tape of 0s and 1s, together with a tape head that moves back and forth, reading and writing. Given a Turing machine M, and given the input $y = y_1, \ldots, y_r$ that's written on its tape, the entire future history of the machine is determined. That means we can draw what is called a *tableau*, which shows the entire history of M's computation at a single glance. Each row of a tableau indicates the state of the Turing machine, the value of the tape, and the position of the Turing machine on the tape.

Time	State	Tape					
6	[H]	0	1	1	1	1	0
5	B	0	1	[1]	1	1	0
4	A	0	[0]	1	1	1	0
3	B	0	0	[0]	1	1	0
2	A	0	0	0	[1]	1	0
1	B	0	0	0	1	[0]	0
0	A	0	0	0	[0]	0	0

Figure 2: Example tableau for a Turing machine that halts after 6 steps.

The tableau for SAT will have T time steps, where T is some polynomial in n, and *because* it only has T time steps, we only ever need to worry about T tape squares from side to side. This means that the whole tableau is of polynomial size. And let's say that the machine accepts if and only if at the very last step some tape square T_i has a 1.

The question that interests us is whether there's *some* setting of the "solution bits" $y = y_1, \ldots, y_r$ that causes M to accept.

How can we create a formula that's satisfiable if and only if such a setting exists? We should certainly include $y = y_1, \ldots, y_r$ as variables of the formula. But we're going to need a whole bunch of additional variables. Basically, for every yes-or-no question you could possibly ask about this tableau, we're going to define a Boolean variable representing the answer to that question. Then we're going to use a slew of Boolean connectives to force those variables to relate to each other in a way that mimics the actions of the Turing machine.

- For all i, t, let $x_{t,i} = 1$ iif at step t, the ith tape square is set to 1.
- Let $p_{t,j} = 1$ iif at step t the Turing machine head is at the ith square.
- Let $s_{t,k} = 1$ iif at step t the Turing machine is in state k.

Next, look at each "window" in the tableau, and write propositional clauses to make sure the right thing is happening in that window.

- First, $x_{1,j} = y_j$ for all j from 1 to r, and $x_{1,j} = 0$ for all $j > r$.
- $p_{t,j} = 0 \Rightarrow x_{t+1,j} = x_{t,j}$. This says that if the tape head is somewhere else, then the jth square is unchanged. We need such a clause for every t and j.
- $s_{t,k} = 1 \wedge p_{t,j} = 1 \wedge x_{t,j} = 1 \Rightarrow s_{t+1,k\prime} = 1$. This says that if, at time t, the machine is in state k and at tape square j, and 1 is written on that tape square, then at the next time step, it will be in state $k\prime$. We'll also need clauses to make sure that $\forall l \neq k\prime : s_{t+1,l} = 0$
- ... and so on.

We need one final clause to make sure that the machine actually accepts.

- $x_{T,0} = 1$

Finally, we string all of these clauses together to produce a humongous—but still polynomial-size!—Boolean formula F, which is consistent (i.e., satisfiable) if and only if there's some $y = y_1, \ldots, y_r$ that causes M to accept.

So, *supposing* we had a polynomial-time algorithm to decide whether a Boolean formula is satisfiable, we could use that algorithm to decide in polynomial time whether there is an input that causes the Turing machine M to accept. In fact, the two problems are equivalent. Or to put it another way, **SAT is NP-complete**. That is the Cook-Levin Theorem.

5.1 Special-case: 2SAT

The above proof doesn't preclude that *special cases* of the SAT problem could be solved efficiently. In particular, consider the problem from Lecture 2 where you have clauses consisting of only two literals each. This problem is called 2SAT, and as we showed in back in Lecture 2, 2SAT is in P. If your Boolean formula has this special form, then there *is* a way to decide satisfiability in polynomial time.

5.2 Special-case: 3SAT

In contrast to 2SAT, let's consider the problem where each clause has *three* literals, or 3SAT. It turns out that 3SAT is NP-complete; in other words, this special case of SAT encodes the entire difficulty of the SAT problem itself. We'll go over the proof for this in Lecture 10.

Appendix

Kurt Gödel's letter to John von Neumann

To set the stage: Gödel is now a recluse in Princeton. Since Einstein died the year before, Gödel barely has anyone to talk to. Later he'll starve himself because he thinks his wife is trying to poison him. Von Neumann, having previously formalized quantum mechanics, invented game theory, told the US government how to fight the Cold War, created some of the first general-purpose computers, and other great accomplishments, is now dying of cancer in the hospital. Despite von Neumann's brilliance, he did not solve the problem that Gödel presented in this letter, and to this day it remains one of the hallmark unsolved problems in theoretical computer science.

Text of Gödel's letter removed due to copyright restrictions
See: http://weblog.fortnow.com/2006/04/kurt-gdel-1906-1978.html