CS 3510 Algorithms 3/30/2023

Lecture 16: Satisfiability

Lecturer: Abrahim Ladha *Scribe(s): Joseph Gulian*

As we discussed last lecture, to prove a problem B is NP-Complete, we must show that for some other NP-complete problem A that $A \leq_P B$ and we must show that B is in NP by providing a verifier. This works because we're showing B is at least as hard as some other NP-Hard problem but it's hardness is bounded by NP, making the problem NP-Complete. However, we have yet to cover any problems that are actually NP-Hard, you could use to show another problem is NP-Hard, meaning you'd have to prove NP-Hard without a reduction. Instead, we'll be giving you an NP-Hard problem today which you can use to prove other problems. [1]

1 Sat

Before we get to the problem, let's define some terms for satisfiability problems. A variable is like a variable in math like a, b, c, d or x_1, x_2, x_3, x_4. They may only take on boolean values 0 or 1. A literal is a variable or the logical not of a variable $a, \neg b, x_1, \neg x_2$ A clause is an $\vee$ of many literals like $a \vee b \vee \neg c$. Conjunctive-Normal-Form (CNF) is an $\wedge$ of many clauses like $(a \vee b \vee \neg c) \wedge (a \vee \neg b \vee c)$.

For Sat, the input is in CNF, and the solution is valid if it is a satisfying assignment to the variables in the input. For instance if a, c were true, if b, c were true or even just if a were true, that would be a satisfying assignment.

Can we come up with an algorithm to solve this? We could brute-force try all the possibilities, or we can do some fancy estimation stuff in polynomial time. However finding a definite answer in polynomial time is hard. Then, can we show it's in NP-Hard? To show this we need to show that solving Sat will solve all NP problems. That seems like a lot of proofs we don't have time for in this class. However it's actually only one proof and it was already proven by Cook-Levin. [2]

We can show that it's NP though. We'll take the assignments as a witness and check that the CNF expression evaluates to true. Since Sat is NP and NP-Hard, and NP-Complete is the intersection of NP and NP-Hard, Sat is NP-Complete. What does this mean again? Solving Sat will give a polynomial time algorithm for all problems in NP.

This may seem like a pretty esoteric problem, but as it turns out it's very useful. However as it turns out, we'll using Sat for problems we discuss in future lectures. Even more so, many real world problems rely on solving something which can be mapped to Sat; think program verification or optimal compilation. To make these problems tractably solvable, we have Sat-solvers which even with exponential time are tractable on small inputs.

[1] Obviously try to understand the reductions, but it's just as important you understand the problems in this unit because you need problems you can use in your reductions.

[2] We don't teach Cook-Levin or care that you know it, but if you're curious you could read chapter 7 of Michael Sipser's excellent "Introduction to the Theory of Computation".

2 3SAT

3SAT is a lot like SAT; the only difference is that clauses are limited to at most three literals. You may immediately come to the conclusion that this is less expressive than SAT, but as it turns out, it's exactly the same. That's to say, any SAT problem has a corresponding 3SAT problem and any 3SAT problem has a corresponding SAT problem. The latter is really easy to understand, so we'll cover that first. If you have a 3SAT problem, it is already a SAT problem because SAT can have clauses of any size including size three or less.

The proof that SAT can be expressed in 3SAT is a little more interesting, but before that let's express why we're interested. We want to show that 3SAT is NP-COMPLETE because this would mean that it can solve all problems in NP. This is very interesting and it implies that the problem 3SAT has some special power. Interestingly, we can use it to prove other problems are NP-COMPLETE.

Clauses in SAT are of the form $(x_1 \vee x_2 \vee ... \vee x_{k-2} \vee x_{k-1} \vee x_k)$. We could split this up into two clauses A and B (for example) by using a new variable: z. We want to create some implications such that $\neg A \implies B$ and $\neg B \implies A$, or in English, if there's no true value in A, there must be a true value in B and vice versa. The $\wedge$ of these two clauses would then have the same effect of one clause i.e., they will both be true if there's one true value. So how do we use our slack variable to have this effect? In one of clauses we have the literal z and in the other clause we have the literal $\neg z$. This transformation would decompose the above clause into the following: $(x_1 \vee x_2 \vee ... \vee x_{k-2} \vee z) \wedge (\neg z \vee x_{k-1} \vee x_k)$. Notice that this creates two clauses with $k-1$ literals and 3 literals. You repeat this until all clauses greater than three have been reduced to many clauses of size at most three.

We need to show that this reduction correctly transforms the problems though, and we'll do this in two ways: the function maps valid instances in one problem to valid instances in another problem, and the function maps invalid instances in one problem to invalid instances in another.

If $\varphi \in$ SAT then there exists a satisfying assignment for each clause. So atleast one of $x_1, ..., x_k$ is on. So there is a choice of z to have both of our new clauses be on and we see $\varphi' \in$ 3SAT

If $\varphi \notin$ SAT then atleast one clause was unsatisfied, suppose it was this one. Then all of $x_1 = ... = x_k = 0$. So then there is no assignment to z to satisfy both clauses so $\varphi' \notin$ 3SAT.

Then we know that the function correctly maps the two problems SAT and 3SAT.

We also need to show that 3SAT is in the class NP. We could simply take the assignments as a witness, plug them in and evaluate. This is in P, so 3SAT is in NP, meaning it's also in NP-COMPLETE (because we formerly showed it was in NP-HARD).

3 KSAT

Let us talk about some other variations of SAT. You should be able to trivially come up with an algorithm for 1SAT. You can simply check for a contradiction, if one doesn't exist there is a solution (assign all the variables such that the literals are true), if there is a contradiction, it can't be done. We'll skip 2SAT for now, and move onto 4SAT (obviously we just proved 3SAT is NP-COMPLETE). You should also have some idea of how to prove that 4SAT is NP-COMPLETE.

Is 2SAT NP-COMPLETE? It may not seem obvious right now, but it actually is in the class P. [3] With sufficient effort, you could likely develop an algorithm to solve 2SAT with the information you've learned in this class.

4 CIRCUITSAT

CIRCUITSAT is an interesting problem where we can use any simple logic gates: AND, OR, and NOT. We can trivially show CIRCUITSAT is in NP by taking a valid assignment to be the witness and checking that it gives a valid output. We can also show that this problem is NP-HARD by performing a reduction with 3SAT. [4] Given any 3SAT problem, we can turn it into a CIRCUITSAT problem by applying the corresponding logic gates to the corresponding 3SAT functionality. If a literal has $\neg$, use the NOT gate. OR multiple literals together and AND the clauses together. Since the expressions are the same, the function is valid. This would imply CIRCUITSAT is NP-HARD, making it NP-COMPLETE since its also NP.

[3] I mean who knows, maybe 3SAT is in P as well.

[4] Often, but not always, 3SAT is an easier choice than SAT for reductions simply because you don't need to worry about arbitrarily large clauses.