CS 4510 Automata and Complexity 3/27/2024

Lecture 19: In and around NP

Lecturer: Abrahim Ladha *Scribe(s): Michael Wechsler*

This lecture could also be titled "The Cook-Levin Theorem and Ladner's Theorem".

1 Reductions

What's the point of intractability and NP-completeness?[1] Suppose you are given a task, and asked to produce an algorithm for some problem, but you can't. So, you then try to prove the problem is intractable or unsolvable. This is pretty hard to do in practice, but it can happen. More likely, you can prove the problem was NP-complete. That elevates it to a special club: a class of problems all as hard as each other. A fast algorithm for one would imply a fast algorithm for all, and that $\mathsf{P} = \mathsf{NP}$. There are thousands of such problems across many domains.

Definition 1.1. We say for two languages $A, B \subseteq \Sigma^*$ that A is polytime reducible to B ($A \leq_p B$) if there is a function, f, which is computable (i.e. halts on all inputs) in polytime with

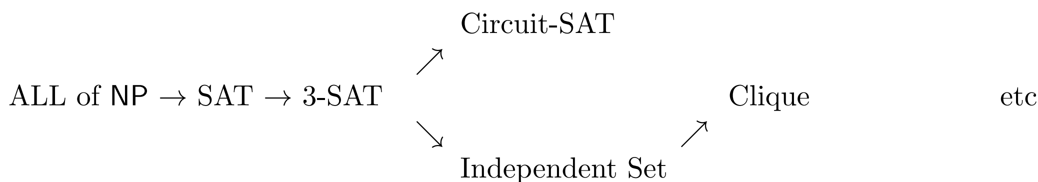
$$w \in A \iff f(w) \in B$$

If $A \leq_p B$, A lower bounds B and B upper bounds A. It is analogous to many-one reductions ($\leq_m$), which are computable at all, while these are required to be computable in polytime. $\leq_m$ can be used to prove problems are as solvable/unsolvable as each other, while $\leq_p$ can be used to prove problems are as easy or hard as each other. Note immediately that if $A \leq_p B$ and $B \in \mathsf{P}$ then $A \in \mathsf{P}$.

2 NP-completeness

We say language B is NP-complete if $B \in \mathsf{NP}$ and $\forall L \in \mathsf{NP}$ that $L \leq_p B$. You may also prove a language to be NP-complete much easier by using the transitivity of the $\leq_p$ relation. Choose some known NP-complete language, A, and prove $B \in \mathsf{NP}$ and $A \leq_p B$. Cook and Levin proved (independently) for us that SAT is NP-complete, which means $\forall L \in \mathsf{NP}$, $L \leq_p$ SAT.

By the web of reductions, we have thousands of other NP-complete problems.



[1] see the attached drawings of Garey and Johnson

$\searrow$
Vertex Cover

This chain can go on and on, and contains cycles.[2] These reductions only work if there is known NP-complete problem, which we are going to prove. A reduction is just a tranformation from one language to another which preserves correctness. The Cook-Levin Theorem proves for every language in NP, there is a reduction to SAT. We will do it generically, for any language in NP.

3 SAT

Recall the definition of SAT:

- Variable: one of $x_1,\ x_2,\ \ldots,\ x_l$
- Literal: one of $x_1,\ x_2,\ \ldots,\ x_l,\ \neg x_1,\ \neg x_2,\ \ldots,\ \neg x_l$. A variable or its negation.
- Clause: an OR of literals, such as $(x_1 \vee \neg x_2 \vee x_3)$
- CNF Formula: an AND of clauses, such as $(x_1 \vee x_2) \wedge (\neg x_3 \vee x_1)$
- Assignment: a selection of $x_1,\ x_2,\ \ldots,\ x_l \in \{0, 1\}$. We say an assignment is satisfying if when you plug in $x_1,\ x_2,\ \ldots,\ x_l$ into a CNF formula that $\phi = 1$.

$$\text{SAT} = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable CNF}\}$$

SAT is the language of all satisfiable formulas. CNFs are surprisingly expressive. Usually in real-world constraint problems, you have a set of constraints and must satisfy all of them, but there may be more than one way to satisfy each. For example,

$$(x_1 \vee \neg y_1) \wedge (\neg x_1 \vee y_1) \wedge \ldots \wedge (x_n \vee \neg y_n) \wedge (\neg x_n \vee y_n)$$

is true if and only if $x_1 = y_1,\ \ldots,\ x_n = y_n$. $x = x_1 \ldots x_n$ and $y = y_1 \ldots y_n \Rightarrow x = y$. This formula is satisfiable if and only if $x = y$. This is a CNF for string equality.

4 Cook-Levin Theorem

Theorem 1 (Cook-Levin Theorem)**.** SAT is NP-complete

Proof. Obviously SAT $\in$ NP since there is a verifier, $V(\phi, c)$, that checks if c is a satisfying assignment to ϕ in polytime.

We want to prove that $\forall L \in \mathsf{NP}$, $L \leq_p$ SAT. Let $L \in \mathsf{NP}$. Then there exists a nondeterministic polytime machine, N, such that N accepts $w \iff w \in L$. We give a reduction $f(w) = \Phi$ such that Φ is satisfiable only when N accepts w. This will allow us to conclude

[2] I hope you recall some of these reductions from the unit in your algorithms course. You should have done many reductions, but all were conditional on the assumption that SAT was NP-complete. Here, we finally prove it.

$w \in L \iff \Phi \in SAT$. The reduction essentially outputs a formula to simulate N on w and it only be satifiable if N accepted w.

Consider the computation history for N on w. We encode the computation history of the machine into a formula, so that the only satisfying assignment is the accepting computation history of the machine, which can only exist if N accepts w. The idea is conceptually simple but has many of little details. Since $L \in \mathsf{NP}$, N is at most polytime, say n^t, and is then also polyspace, say n^s. Take the sequence of configurations of an accepting computation history $C_0,\ C_1, \ldots$ and line them up in a table like so

#	q_0	1	1	1	␣	#
#	0	q_0	1	1	␣	#
#	0	0	q_0	1	␣	#
#	0	0	0	q_0	␣	#
#	0	0	0	␣	q_a	#

Note: There are n^t rows and $n^s + 2$ish columns

The dimension of the tableau is time by space $= n^t \times n^s$, making it polynomial sized. We encoe the table into a CNF. We will create a CNF formula, Φ, to loop over the table and check its correctness. We can say such a table exists and is correct if exactly and only four conditions are met. We simulate each of the four conditions with a CNF.

$$\Phi = \Phi_{cell} \wedge \Phi_{start} \wedge \Phi_{move} \wedge \Phi_{accept}$$

- $\Phi_{cell} = 1 \iff$ exactly one symbol is in each cell of the table
- $\Phi_{start} = 1 \iff$ first row is the initial configuration
- $\Phi_{move} = 1 \iff$ the $i+1^{th}$ row is a valid configuration following the application of δ of N to the ith row.
- $\Phi_{accept} = 1 \iff$ there is an accepting configuration in the table

Let $x_{i,j,s}$ be the variables with $1 \leq i \leq n^t,\ 1 \leq j \leq n^s,\ s \in Q \cup \Gamma \cup \{\#\}$ where

$$x_{i,j,s} = 1 \iff \text{cell}[i,j] = s$$

$x_{i,j,s}$ means symbol s is in cell $[i,j]$. Let $C = Q \cup \Gamma \cup \{\#\}$.

- $\Phi_{cell} = 1 \iff$ exactly one symbol is in each cell of the table. For example, we want a relationship like $x_{i,j,a} = 1 \implies x_{i,j,b} = 0$. We can do this by ANDing clauses which make sure atleast one symbol is on for each cell, and no more than one is on for each cell. This ensures each element of the table has exactly one symbol.

$$\Phi_{cell} = \bigwedge_{\substack{1 \leq i \leq n^t \\ 1 \leq j \leq n^s}} \left[\Big(\bigvee_{s \in C} x_{i,j,s} \Big) \bigwedge \Big(\bigwedge_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \Big) \right]$$

$\bigwedge\limits_{\substack{1\le i\le n^t\\1\le j\le n^s}}$: double for loop over the entire two dimensional table

$\bigvee\limits_{s\in C}$: guarantees at least one symbol is in each cell

$\bigwedge\limits_{\substack{s,t\in C\\s\neq t}}$: guarantees no more than one symbol is in each cell

We are essentially using the syntax of SAT to write a program to check correctness of our table. Its not essential to understand how this "programming language" works. Its more important to understand that this was even possible.

- $\Phi_{start} = 1 \iff$ first row is the initial configuration of N on w with appropriate space.

  $$\Phi_{start} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge x_{1,n+3,\llcorner\lrcorner} \wedge \ldots \wedge x_{1,n^s-1,\llcorner\lrcorner} \wedge x_{1,n^s,\#}$$

  To satisfy Φ_{start}, the corresponding table must have the first row of our desired configuration

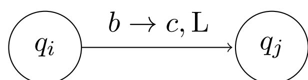
- $\Phi_{accept} = 1 \iff$ there is an accepting configuration in the table

  $$\Phi_{accept} = \bigvee_{\substack{1\le i\le n^t\\1\le j\le n^s}} x_{i,j,q_a}$$

  Loop over the entire table to make sure q_a symbol exists somewhere

- $\Phi_{move} = 1 \iff$ the $i+1^{th}$ row is the $C_{i+1^{th}}$ configuration, following the ith row

  Φ_{move} is the hardest one. We want it to enforce that each row follows the preceding one by only legal moves according to the transition function δ of N. With the first initial configuration enforced, we want Φ_{move} to enforce row two is the second configuration and so on. The way Φ_{move} will work is check every 2×3 window of the table and determine if it's a legal 2×3 window. For example, for transitions like



a	q_i	b
q_j	a	c

this is a legal window. There are other legal 2×3 windows like

a	b	c
a	b	c

a	a	q_i
a	a	b

#	a	b
#	a	q_j

Using the transition function, you may enumerate all possible legal windows, and there are only finitely many of them. For this table, I have dotted a few windows near the head. If the whole table is legal, the windows near the head are the only ones which aren't copy windows. Convince yourself that the $i+1^{th}$ row follows from the ith row if and only if every 2×3 window is legal. These are equivalent conditions.

$$\Phi_{move} = \bigwedge_{\substack{1 \leq i \leq n^t \\ 1 \leq j \leq n^s}} [\text{the } (i,j) \text{ window is legal}]$$

By legal, we mean according to δ of N.

$$\Phi_{move} = \bigwedge_{\substack{1 \leq i \leq n^t \\ 1 \leq j \leq n^s}} [\bigvee_{\substack{a_1,\ldots,a_6 \\ \text{is legal}}} (x_{i,j-1,a_1} \wedge \ldots \wedge x_{i+1,j+1,a_6})]$$

$\bigwedge_{\substack{1 \leq i \leq n^t \\ 1 \leq j \leq n^s}}$: double for-loop over two dimensional table, checking all 2×3 windows

$\bigvee_{\substack{a_1,\ldots,a_6 \\ \text{is legal}}} (x_{i,j-1,a_1} \wedge \ldots \wedge x_{i+1,j+1,a_6})$: checks if window i,j is legal

We finish by construction of

$$\Phi = \Phi_{cell} \wedge \Phi_{start} \wedge \Phi_{move} \wedge \Phi_{accept}$$

Note: Φ is satisfiable only if:

1. Each cell of the table contains exactly one symbol
2. The first row is a start configuration
3. The $(i+1)^{th}$ row is the $C_{i+1^{th}}$ configuration following the ith row as the C_i^{th} configuration
4. One of the configurations is accepting

So Φ is satisfiable only if an accepting computations history of N on w exists, which can only happen if there is a computation of N on w so $w \in L \iff \Phi \in \text{SAT}$

Now we argue that this reduction takes polytime to compute. Note that for a polynomial sized table, each of the subformulas also took polynomial time to construct so the computation to build Φ takes polytime. We construct Φ by just a few for-loops. We observe $L \leq_p$ SAT. Since SAT $\in \mathsf{NP}$ and $\forall L \in \mathsf{NP}$, $L \leq_p$ SAT, we conclude SAT is NP-complete. □

4.1 Importance of this Finding

First, consider how the simplicity of the proof relied on the simplicity of the Turing machine. Surely it could be done for a different computational model, but the proof would also be far more complex.

SAT is not the only language that could be proved as a genesis NP-complete problem. Sipser and CLRS both include a proof by a similar construction that CircuitSAT is NP-complete. Levin originally proved a kind of tiling problem. Cook proved not SAT necessarily, but something like tautologies are NP-complete. You may remark this proof is similar to the proof that PCP was undecidable, in the sense we encoded the accepting computation history into the language of the problem.

Now that we have proven SAT is NP-complete, we may prove many other languages are NP-complete, not by repeating the proof, but by a simple reduction. For example, if you prove 3SAT $\in \mathsf{NP}$ and SAT $\leq_p$ 3SAT, then since we proved $\forall L \in \mathsf{NP}$ that $L \leq_p$ SAT, we can use transitivity. $L \leq_p \text{SAT} \leq_p \text{3SAT} \implies L \leq_p \text{3SAT}$. The reduction reuses and transforms the proof, rather than redoing it.

As a final remark on this reduction. Note if we repeated it with a polytime deterministic verifier instead of a nondeterministic polytime machine, it would map from the witness of the verifier to the satisfying assignment of the formula. This reduction can even be used to map witnesses to witnesses.

The complexity of an entire class, NP, can be reduced to the complexity of this single simply defined problem.

Theorem 2. $\text{SAT} \in \mathsf{P} \iff \mathsf{P} = \mathsf{NP}$

Proof. To prove, recall $\forall L \in \mathsf{NP}$ that $L \leq_p$ SAT. So if SAT $\in \mathsf{P}$, then there is a polytime algorithm for SAT. Since every $L \in \mathsf{NP}$ is polytime reducible to SAT, combining this reduction plus the polytime algorithm for SAT is a polytime algorithm for L. So $L \in \mathsf{P}$, but since L is any language in NP, we see $\mathsf{NP} \subseteq \mathsf{P}$. Since we know $\mathsf{P} \subseteq \mathsf{NP}$, we conclude $\mathsf{P} = \mathsf{NP}$. The reverse is true since we proved SAT $\in \mathsf{NP}$, so $\mathsf{P} = \mathsf{NP} \implies \text{SAT} \in \mathsf{P}$. □

We do not believe SAT has a polytime algorithm. We don't even believe SAT has a quasi-polytime algorithm.

5 Ladner's Theorem

Not all languages in $\mathsf{NP} \setminus \mathsf{P}$ are NP-complete if $\mathsf{P} \neq \mathsf{NP}$. We will prove it shortly. Factoring is a candidate for an NP-intermediate problem. It is (believed) not to be in P, but has

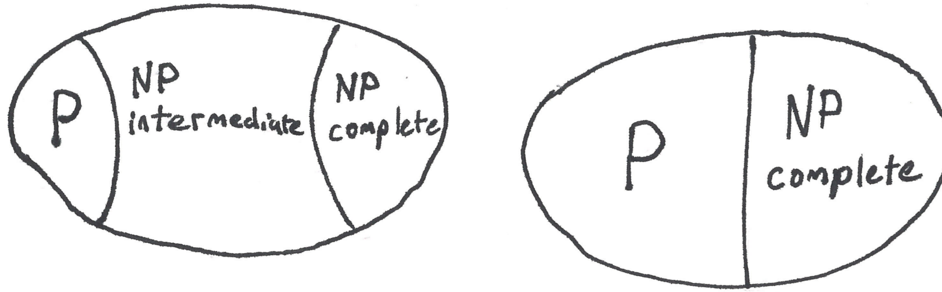


Figure 1: Ladner's Theorem proves the left case and disproves the right case

sub-exponential time algorithms. The general number field sieve has a runtime for factoring $\in TIME(o((1+\varepsilon)^n))\ \forall \varepsilon > 0$. We believe that factoring cannot be in P, otherwise, cryptography doesn't exist. We say a function is quasi-polynomial if it is super polynomial, yet subexponential. Such functions do exists, and algorithms exists with quasi-polynomial run-time.

5.1 Exponential Time Hypothesis

The hardness of SAT can be formalized as an assumption, ETH = Exponential Time Hypothesis. Essentially, ETH states SAT cannot be solved in subexponential time. It is a stronger assumption than $\mathsf{P} \neq \mathsf{NP}$ since it implies $\mathsf{P} \neq \mathsf{NP}$ but also other things. If you assume ETH, you are assuming SAT has no $2^{o(n)}$ time algorithm. In certain proofs, if we assume ETH instead of $\mathsf{P} \neq \mathsf{NP}$, it will make some proofs easier.

5.2 The proof

We prove Ladner's Theorem: If $\mathsf{P} \neq \mathsf{NP}$ then there exists languages which:

- are not in P
- are in NP
- are not NP-complete

We are proving that if $\mathsf{P} \neq \mathsf{NP}$ then the NP-intermediate languages exist. If you could prove these exists unconditionally, then of course you found a language in $\mathsf{NP} \setminus \mathsf{P}$, and would prove $\mathsf{P} \neq \mathsf{NP}$. The original proof was clearly diagonalization simultaneously against all polytime algorithms and all polytime reductions. Here, we do a proof by a kind of padding argument. We will assume ETH instead of $\mathsf{P} \neq \mathsf{NP}$ to get an easier proof with the same ideas.

Theorem 3. (Weaker Ladner's Theorem) If SAT requires exponential time then there exists languages which are in NP-intermediate.

Proof. Assume ETH. There is no sub-exponential time algorithm for SAT, that SAT $\notin TIME(2^{o(n)})$. Recall that we measure the run-time of an algorithm as a function of the input size. If we take a reasonable problem, and then just pad on a bunch of stuff, we can

say an algorithm is sub-exponential in the padded size, even if it wasn't sub-exponential in the true size. Our NP-intermediate language will simply be SAT padded SAT by a quasi polynomial amount.

$$L = \{\langle \phi, 1^{2^{\sqrt{|\phi|}}} \rangle \ \mid \phi \in \text{SAT}\}$$

- First we show that $L \in \mathsf{NP}$. Our witness is the assignment, same as SAT. Our verifier V on input $\langle w, c \rangle$ checks if w is of the form $\langle \phi, 1^{2^{\sqrt{|\phi|}}} \rangle$, doing some math to count the padding. Then it checks if c is a satisfying assignment for ϕ. If it is then it accepts. In terms of the size of the input, the verifier V takes polynomial time, so we see that $L \in \mathsf{NP}$

- Next we show $L \notin \mathsf{P}$. Suppose it was. Then there exists an algorithm, A, to decide L in time polynomial in the size of the input. A runs in time $O((n + 2^{\sqrt{n}})^k)$ for some k, where n is the size of just the formula and not the padding. Note that $n + 2^{\sqrt{n}}$ is the size of the input. We give a sub-exponential time algorithm, A', for SAT.

Algorithm 1 A'

on input ϕ
build $\langle \phi, 1^{2^{\sqrt{|\phi|}}} \rangle = y$
run $A(y)$

A' takes $O(2^{\sqrt{n}})$ to write down y, then $O((n + 2^{\sqrt{n}})^k)$ to run A. So A' decides SAT in time in $O(2^{\sqrt{n}}) + O((n + 2^{\sqrt{n}})^k) = 2^{O(\sqrt{n})} = 2^{o(n)}$. This sub exponential time algorithm for SAT violates our assumption of ETH, so $L \notin \mathsf{P}$.

- Now we show L is not NP-complete. The proof idea is that if it was, there is a reduction for it, such a reduction could be used to solve SAT too fast, violating ETH. Assume to the contrary that L is NP-complete. Then there exists a polytime computable reduction such that SAT $\leq_p L$. This reduction function, f, works such that $f : \psi \to \langle \phi, 1^{2^{\sqrt{|\phi|}}} \rangle$ and $\psi \in SAT \iff \langle \phi, 1^{2^{\sqrt{|\phi|}}} \rangle \in L$, where ψ, ϕ may be different. Since our f is polytime, there exists some k such that $|f(\psi)| = n^k$. Any polynomial time algorithm (here, a reduction) can only produce a polynomial-sized output. It take time to write that output down. Here $|\psi| = n$, the size of the input. The reduction outputs $\langle \phi, 1^{2^{\sqrt{|\phi|}}} \rangle$. Since a poly time reduction must have a polysized output, the only way that this output could be polynomial in terms of n if if ϕ must be small enough such that $2^{\sqrt{|\phi|}} \leq |f(\phi')| = n^k$, so

$$2^{\sqrt{|\phi|}} \leq n^k \Rightarrow \sqrt{|\phi|} \leq k \log n \implies |\phi| \leq (k \log n)^2$$

or that $|\phi| << n$, $o(n)$.

Since $|\phi|$ is much smaller than n, than $|\psi|$, its faster for us to brute force check assignments of ϕ than of ψ. To see if $\psi \in$ SAT, perform the polytime reduction and try all assignments of ϕ. This will take time $2^{(k \log n)^2} = 2^{o(n)}$, violating ETH.

Therefore, we conclude that assuming ETH implies there exists a language L such that $L \notin \mathsf{P}$, $L \in \mathsf{NP}$, but L is not NP-complete. So the class NP-intermediate exists. □

Note that the difficulty of the proof changes if we have to use the assumption $\mathsf{P} \neq \mathsf{NP}$ instead of ETH. We reached a contradiction twice using the fact that SAT had a $2^{o(n)}$ time algorithm. If we had to do the full proof, we would have to show that SAT had a polytime algorithm, and not just a sub-exponential one. This proof can even be further strengthed to show that NP-intermediate contains a countably infinite hierarchy with languages $..., L_i, L_{i+1}, ...$ such that $L_i \leq_p L_{i+1}$ but $L_{i+1} \not\leq_p L_i$.