

Lecture Notes 6:**Padding, Ladner's Theorem, Relativization****Reading.**

- Arora-Barak § 3.4, 4.1

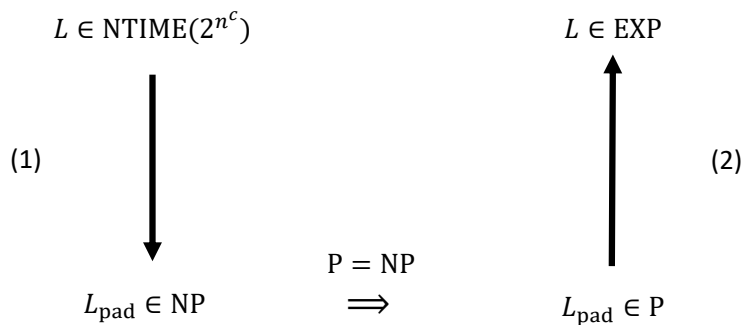
Last time: Decision vs. search, hierarchy theorems

1 Padding

Padding is a useful technical trick for showing (among other things) that complexity class collapses “scale up” with more resources, or equivalently, that class separations “scale down.” It’s best illustrated with an example. (Theorem 2.22 in Arora-Barak.)

Theorem 1. *If $P = NP$, then $EXP = NEXP$.*

To prove this, it suffices to show that if $L \in \text{NTIME}(2^{n^c})$ for some constant c , and $P = NP$, then $L \in \text{EXP}$. The way we’re going to do this is to “pad” the language L to construct a new language L_{pad} that is easier to solve relative to its input length. Schematically:



The construction of this padded language is as follows.

$$L_{\text{pad}} = \{x1^{2^{|x|^c}} \mid x \in L\}.$$

Claim 1. If $L \in \text{NTIME}(2^{n^c})$, then $L_{\text{pad}} \in \text{NP}$.

Proof. Let N be an NTM deciding L in time $O(2^{n^c})$. Consider the following NTM N' deciding L_{pad} : On input y :

1. Check if $y = x1^{2^{|x|^c}}$ for some x . If not, reject.
2. Run N on input x and return its answer.

This runs in time $O(2^{|x|^c}) = O(|y|)$, which is polynomial in the input length. $\square$

Claim 2. If $L_{\text{pad}} \in \mathbf{P}$, then $L \in \mathbf{EXP}$.

Proof. Let M decide L_{pad} in (deterministic) time $O(n^k)$. Consider the following TM M' deciding L : On input x :

1. Construct $y = x1^{2^{|x|^c}}$.
2. Run M on input y and return its answer.

This runs in time $O(|y|^k) = O(2^{k|x|^c}) = O(2^{|x|^{c+1}})$, which is exponential in the input length. $\square$

2 Ladner's Theorem

Ladner's Theorem addresses the following basic question about the structure of $\mathbf{NP}$: Are there languages that are neither in $\mathbf{P}$ nor $\mathbf{NP}$ -complete? The answer is that there are, assuming $\mathbf{P} \neq \mathbf{NP}$. (In fact, there's an infinite hierarchy of classes strictly containing $\mathbf{P}$ and strictly contained in $\mathbf{NP}$.)

Theorem 2. If $\mathbf{P} \neq \mathbf{NP}$, there exists a language L such that $L \notin \mathbf{P}$ and L is not $\mathbf{NP}$ -complete.

These notes contain two brief proof sketches. The first follows the presentation in Arora-Barak (which they credit to Impagliazzo).

Proof 1 of Theorem 2. The basic idea is to define a padded version of SAT that reduces its hardness in a controlled manner. Specifically, for a function $T(n)$, define

$$\text{SAT}_T = \{\varphi 1^{T(|\varphi|)} \mid \varphi \text{ is satisfiable}\}.$$

The hardness of SAT_T depends on the growth rate of T . As two extreme examples:

1. If T is polynomial, then SAT_T is $\mathbf{NP}$ -complete. The poly-time reduction from SAT to SAT_T is just $f(\varphi) = \varphi 1^{T(|\varphi|)}$.
2. If $T(n) = 2^n$, then $\text{SAT}_T \in \mathbf{P}$. This is because, given $\varphi 1^{T(|\varphi|)}$, an algorithm can just brute force over all satisfying assignments in time $2^{|\varphi|}$, which is now linear in the input length.

The proof of Ladner's Theorem (roughly) makes the following choice for $T(n)$: It is the smallest function T such that SAT_T can be solved in time $T(n)$. [This self-referential definition is tricky to formalize, and it's not exactly this. The formal definition uses diagonalization.]

Now there are two cases depending on what T turns out to be.

1. If $T(n) = \text{poly}(n)$, then SAT_T is $\mathbf{NP}$ -complete, as we said above. But now there's a $T(n)$ -time algorithm solving this $\mathbf{NP}$ -complete problem, so $\mathbf{P} = \mathbf{NP}$.

2. If $T(n) = n^{\omega(1)}$ (e.g., $T(n) = n^{\log n}$), our goal is to show that SAT_T is **NP**-complete $\implies \mathbf{P} = \mathbf{NP}$. So suppose SAT_T is **NP**-complete. Then $\text{SAT} \leq_p \text{SAT}_T$ via some $O(n^k)$ -time computable reduction f . We'll use this to give a poly-time algorithm for SAT itself. Let φ be a Boolean formula of length n . First, apply the reduction f to get a SAT_T instance $\varphi_1 1^{T(|\varphi_1|)}$. Since T is a growing function, $|\varphi_1| \ll |\varphi|$; say for concreteness, that $|\varphi_1| \leq \sqrt{|\varphi|}$. We can repeat this process $\log \log n$ times until we get down to a formula $\varphi_{\log \log n}$ of constant size that is satisfiable iff φ is; at this point, we can just brute force over candidate assignments.

Proof 2 of Theorem 2. The second proof is based on Ken Regan's exposition of an argument due to Schöning. You may find it more conceptually intuitive, and it makes the role of diagonalization more readily apparent.

Suspend disbelief for a moment to assume that we can enumerate all languages in **P** as $A_1, A_2, \dots$ and all **NP**-complete languages as $B_1, B_2, \dots$. Our goal is to define a language L that disagrees with every one of these languages on some input.

As a first attempt, consider the following search process enumerating over strings $y \in \{0, 1\}^*$ looking for "bad" instances of these languages:

1. Find the first string y_1 such that $A_1(y_1) \neq \text{SAT}(y_1)$. [Such a string has to exist because A_1 does not decide SAT .] Put $y_1 \in L \iff y_1 \in \text{SAT}$.
2. Continuing the search from y_1 , find the first string y'_1 such that $B_1(y'_1) = 1$. [Such a string has to exist because **NP**-complete languages must be infinite.] Exclude y'_1 from L .
3. Continuing the search from y'_1 , find the first string y_2 such that $A_2(y_2) \neq \text{SAT}(y_2)$. Put $y_2 \in L \iff y_2 \in \text{SAT}$.
4. ... Keep alternating back and forth between the A 's and the B 's.

Thus, we've (partially) defined L so that, for every language $A_1, A_2, \dots, B_1, B_2, \dots$, there is a string whose status in L disagrees with it. Aside from L not being defined on other strings, the main issue is that it's not clear how to place it in **NP**. We'll fix both issues with this construction by recording "time stamps" for when the search process finds each of the bad strings:

1. Set $t'_0 = 0$.
2. Find the first string y_1 such that $A_1(y_1) \neq \text{SAT}(y_1)$. Let t_1 be the number of time steps it took to find y_1 .
3. Continuing the search from 0^{t_1} , find the first string y'_1 such that $B_1(y'_1) = 1$. Let t'_1 be the (total) number of time steps it took to find y'_1 .
4. Continuing the search from $0^{t'_1}$, find the first string y_2 such that $A_2(y_2) \neq \text{SAT}(y_2)$. Let t_2 be the (total) number of time steps it took to find y_2 .
5. ... Keep alternating back and forth between the A 's and the B 's.

Now we define the language L by

$$L(x) = \begin{cases} \text{SAT}(x) & \text{if } \exists i \quad t'_i \leq |x| < t_{i+1} \\ 0 & \text{otherwise.} \end{cases}$$

As before, L still disagrees with every language $A_1, A_2, \dots, B_1, B_2, \dots$ on the special strings y_i, y'_i . But now we can give an **NP**-verifier for L as follows: Use the usual certificate for SAT. On input x , run the search procedure for $|x|$ steps to see what case we're in. If we're in the first case, check the certificate. The really clever feature of this construction is that we do not actually need to complete the search on input x . We just need to run it for the right number of steps to be able to tell what case we're in.

Finally, we have to deal with the issue of enumerating over all poly-time decidable languages and **NP**-complete languages. For **P** languages, we can just enumerate over all pairs consisting of a TM and a polynomial runtime bound, and terminating the TM early when we hit that runtime. For **NP**-complete languages, we can enumerate over tuples consisting of 1) an NTM deciding a language L , 2) a TM representing an alleged reduction f from SAT, and 3) polynomial runtimes for each of these. For each such tuple, we can build a language decided by the following machine: On input x , spend $|x|$ steps exhaustively searching for an input on which f fails to reduce SAT to L . If no failure was found, use L . Otherwise, use the answer to SAT.

3 Limits of Diagonalization

We've seen some very clever diagonalization arguments so far, e.g., for the nondeterministic time hierarchy theorem and Ladner's theorem. Could diagonalization be used to prove a landmark result like $\mathbf{P} \neq \mathbf{NP}$?

There's a formal sense in which the answer turns out to be "no," but to state this, we have to define exactly what we mean by "diagonalization." We'll do so indirectly by examining what properties of the TM model we actually use to carry out these arguments, which turns out to not be very much. Specifically, we need the facts that

- TMs can be encoded by strings and enumerated, and
- There is a universal TM that can efficiently simulate any other TM.

Any proof that uses only these properties will also go through, unchanged, for any other computational model with these properties. One such (family) of models is that of oracle TMs.

Definition 3 (Oracle TM). Let $A \subseteq \{0, 1\}^*$ be a language. An TM with oracle access to A , denoted M , has:

1. An extra read/write "oracle tape."
2. The ability to, in 1 computational time step, obtain the answer to the question "Is $q \in A$?" where q is the string written on the oracle tape.

All of the complexity classes we've seen can be defined in the presence of oracles. For instance,

$$\mathbf{P}^A = \{\text{languages decided by poly-time deterministic TMs with oracle access to } A\}$$

$$\mathbf{NP}^A = \{\text{languages decided by poly-time nondeterministic TMs with oracle access to } A\}$$

So what do oracles do to these classes? Of course, it depends on the oracle.

Example 4. If $A \in \mathbf{P}$, then $\mathbf{P}^A = \mathbf{P}$. To see that $\mathbf{P} \subseteq \mathbf{P}^A$, note that an oracle TM M^A can always ignore its oracle. For the other direction, $\mathbf{P}^A \subseteq \mathbf{P}$, we can simulate each call to the oracle by deciding A in poly-time.

Example 5. $\text{NP} \cup \text{coNP} \in \text{P}^{\text{SAT}}$. (The latter is the class of languages that are Cook-reducible to SAT.)

A proof of a complexity class separation $\text{C}_1 \neq \text{C}_2$ (resp. collapse $\text{C}_1 = \text{C}_2$) relativizes if it continues to hold in the presence of any oracle, i.e., for all oracles A , we have $\text{C}_1^A \neq \text{C}_2^A$ (resp. collapse $\text{C}_1^A = \text{C}_2^A$). Any conjecture that is inherently non-relativizing (i.e., because it's true in the presence of some oracle and false in the presence of another) requires non-relativizing techniques to resolve.

It turns out that the P vs. NP problem can't have a relativizing solution.

Theorem 6. *There exist oracles A, B such that*

$$\text{P}^A = \text{NP}^A, \quad \text{P}^B \neq \text{NP}^B.$$

Proof. First, we construct the oracle A for which $\text{P}^A = \text{NP}^A$:

$$A = \{\langle M, x, 1^n \rangle \mid M \text{ accepts } x \text{ within } 2^n \text{ steps}\}.$$

One can show that A is EXP -complete under poly-time reductions, so in particular, $\text{EXP} \subseteq \text{P}^A$.

Meanwhile, we can also see that $\text{NP}^A \subseteq \text{EXP}$. This is because in exponential time, we can brute-force over all the NP machine's nondeterministic choices, and simulate all of the queries to A in exponential time.

Now let's construct the oracle B for which $\text{P}^B \neq \text{NP}^B$. The first thing we'll actually do is describe the form of the *language* that is in NP^B , but not in P^B :

$$U_B = \{1^n \mid \exists x \in B, |x| = n\}.$$

Claim 1: For every oracle B , we have $U_B \in \text{NP}^B$.

Proof of Claim 1: To decide if $1^n \in U_B$, nondeterministically guess a string $x \in \{0, 1\}^n$ and use the oracle to check if $x \in B$.

Claim 2: There exists an oracle B such that $U_B \notin \text{P}^B$.

Proof of Claim 2: This is the most interesting part of the proof, and perhaps ironically, constructs the oracle B by diagonalizing against all poly-time oracle TMs. Here's the gameplan. Let $M_1, M_2, \dots$ be an enumeration of all TMs. We're going to construct the oracle B in stages:

$$\emptyset = B_0 \subseteq B_1 \subseteq B_2 \subseteq \dots$$

such that for every stage i and every $j > i$, the TM $M_j^{B_j}$ does not decide U_{B_j} within $2^n/10$ steps. Our final oracle will be

$$B = \bigcup_{i=0}^{\infty} B_i.$$

Our stagewise construction will produce each B_i from B_{i-1} . Let n_i be large enough so that $|q| < n_i$ for every oracle query q made by every previous machine $M_j^{B_j}(1^{n_j})$ for $j < i$. Our goal is to construct B_i to force $M_i^{B_i}$ to mess up on input 1^{n_i} .

Run $M_i^{B_i}$ on input 1^{n_i} for $2^{n_i}/10$ steps, answering its oracle queries as follows:

- If q was queried in a previous round, answer consistently with the previous oracle B_{i-1} .
- If q is new, answer no, i.e., $q \notin B_i$.

After $2^{n_i}/10$ steps, there are two possibilities:

- If $M_i^{B_i}(1^{n_i})$ accepts, set $q \notin B_i$ for every $q \in \{0, 1\}^{n_i}$. By the definition of B_i , this makes $1^{n_i} \notin U_{B_i}$, so $M_i^{B_i}$ fails to decide U_{B_i} .
- If $M_i^{B_i}(1^{n_i})$ does not accept, then let $q \in \{0, 1\}^{n_i}$ be an arbitrary string not queried so far. Such a string exists because we stopped the simulation after only $2^{n_i}/10$ steps, so it could have only made this many queries. Set $q \in B_i \implies 1^{n_i} \in U_{B_i}$, which again ensures that $M_i^{B_i}$ does not decide U_{B_i} .

Finally, note that B_j is consistent with B_i on all of M_i 's oracle queries for all $j > i$, so we conclude the stronger requirement that M_i^B does not decide U_B . $\square$

Remark 7. The structure of the construction of B doesn't actually have much to do with the fact that we're separating $\mathbf{P}$ from $\mathbf{NP}$. Really, it just uses the facts that 1) one can decide membership in U_B by making one nondeterministic query to B and 2) making fewer than $2^n/10$ deterministically-chosen queries to B does not suffice. An equivalent interpretation of the construction and analysis of U_B is that computing the OR function on 2^n inputs can be done with one nondeterministic query, but requires decision-tree depth $> 2^n/10$. If you want construct oracles that separate other complexity classes, a good way to do it is to identify the appropriate analogs of decision trees (equivalently, query algorithms) and prove lower bounds against those.

Barriers, more generally. If we're going to prove $\mathbf{P} \neq \mathbf{NP}$, we're going to need techniques that fail to relativize. By now, a few such techniques are available. One is circuit lower bounds, which compile TMs down to Boolean circuits and use more detailed combinatorial information about them. These, unfortunately, are often subject to the Razborov-Rudich "natural proofs" barrier. Another is algebraic techniques, which later in the course we'll see can be used to prove things like the $\mathbf{IP} = \mathbf{PSPACE}$ theorem. These too, however, are subject to the Aaronson-Wigderson "algebrization" barrier. So even if we're still far from proving $\mathbf{P} \neq \mathbf{NP}$, we have some good excuses for it!