CAS CS 535: Complexity Theory

Lecturer: Mark Bun — Fall 2023

Lecture Notes 5:

Hierarchy Theorems, Padding, Ladner's Theorem

Reading.

- Arora-Barak § 3.1-3.3, 4.1.3

Last time: Decision vs. Search

- M runs in time $f(n)$ if it halts within $f(|x|)$ steps for every $x \in \{0,1\}^*$.
- M runs in space $f(n)$ if it visits at most $f(|x|)$ cells of its work tapes for every $x \in \{0,1\}^*$.

Definition 1. For a function $f(n)$, define the complexity classes

$$\mathbf{DTIME}(f(n)) = \{L \subseteq \{0,1\}^* \mid L \text{ is decidable in time } O(f(n))\}.$$

$$\mathbf{SPACE}(f(n)) = \{L \subseteq \{0,1\}^* \mid L \text{ is decidable in space } O(f(n))\}.$$

1 Hierarchy Theorems

Reductions and the theory of $\mathbf{NP}$-completeness give us tools to understand the relative hardness of computational problems. But on their own, they don't give us a way to establish, say, that any of the problems in $\mathbf{NP}$ are themselves actually hard.

The main tool we have for actually separating complexity classes is diagonalization, which we'll first use here to prove hierarchy theorems. A hierarchy theorem says that strictly more of a particular resource lets you compute strictly more things.

We'll start with the deterministic time hierarchy theorem, where the resource is time on a deterministic TM. To state the theorem precisely, we need a slightly annoying definition.

Definition 2. A function $f : \mathbb{N} \to \mathbb{N}$ is time-constructible if there exists a TM computing $f(|x|)$ within $O(f(|x|))$ steps.

Most reasonable functions $f(n) \geq n$ are time-constructible, e.g., $n^{1.5}, n\log n, 10^n$. Some functions that are not time-constructible are $\log n, n^{0.9}$ (too small), Busy-Beaver(n) (too large to be computable).

Theorem 3 (Deterministic Time Hierarchy Theorem)**.** *If f and g are time-constructible and $f(n)\log f(n) = o(g(n))$, then*

$$\mathbf{DTIME}(f(n)) \subsetneq \mathbf{DTIME}(g(n)).$$

That is, while $\mathbf{DTIME}(f(n)) \subseteq \mathbf{DTIME}(g(n))$ (obviously), there exists a language $L \in \mathbf{DTIME}(g(n))$, but $L \notin \mathbf{DTIME}(f(n))$.

For example, if $f(n) = n$ and $g(n) = n \log^2 n$, then because $n \log n = o(n \log^2 n)$ we have that $\mathbf{DTIME}(n) \subsetneq \mathbf{DTIME}(n \log^2 n)$.

Before proving the theorem, I want to draw attention to two apparent deficiencies. One is the "extra" factor of $\log f(n)$; ideally, one would like to be able to separate time classes whenever $f(n) = o(g(n))$. This factor has to do with the best known simulation of arbitrary-tape TMs by 3-tape TMs, which incurs the same logarithmic runtime blowup.

Second is the restriction to time-constructible f and g. This looks like a technicality, but it turns out to be important to the truth of the statement. There's a result called the Borodin-Trakhtenbrot Gap Theorem which implies there is a non-time constructible function f for which $\mathbf{DTIME}(f(n)) = \mathbf{DTIME}(2^{f(n)})$.

As mentioned before, the idea we'll use to prove the time hierarchy theorem is diagonalization. This is the same idea that was implicit in our construction of the undecidable language SNA, so let's spell out that construction in a bit more detail.

Imagine constructing a 2-dimensional grid where rows are indexed by possible input strings x, and columns are indexed by Turing machines encoded by strings α. In each cell (x, α), we write the value of $M_\alpha(x)$.

(Encodings of) Turing machines

Inputs

	α_1	α_2	α_3	α_4	
x_1	~~0~~ 1	1	0	0	…
x_2	1	~~1~~ 0	0	000	
x_3	11	10	~~1~~ 1	0	
x_4	1	0	1	~~1001~~ 1	
$\vdots$					$\ddots$

The language SNA was defined by "flipping the diagonal" to force every TM M_α to make a mistake when run on input $x = \alpha$. That is,

$$\mathsf{SNA} = \{x \mid M_x(x) \neq 1\}.$$

To prove our time hierarchy theorem, we'd like to use the same idea, but

1. The language we obtain by flipping the diagonal should be decidable in $O(g(n))$ time
2. We only need to thwart machines that run in time $O(f(n))$.

As a first attempt to do this, we might try only enumerating over all TMs running in time $O(f(n))$. The problem is that it's undecidable to determine whether a given TM M runs in time $O(f(n))$. So instead, we'll go back to enumerating over all TMs, but stop each simulation early, after only $g(n)$ steps.

Proof. We construct a language $L \in \mathbf{DTIME}(g(n)) \setminus \mathbf{DTIME}(f(n))$ as follows. Construct the TM $D(x) =$

1. Run the UTM $U(x, x)$ to simulate M_x on input x, and stop after the UTM has run for $g(|x|)$ time steps.
2. If U has output 1, return 0. Otherwise, return 1.

Let L be the language decided by D. By construction, $L \in \mathbf{DTIME}(g(n))$.

Now suppose for the sake of contradiction that some TM M decides L in time $O(f(n))$. By the efficiency of the UTM, simulating M to completion on any input x takes time $c_M f(|x|) \cdot \log f(|x|)$ for some c_M depending only on the machine M.

Now this is the part of the proof where some assumptions about how we encode TMs matter. Let us assume that every TM has infinitely many encodings (say, by appending an arbitrary number of trailing 0s). Then the fact that $f(n) \log f(n) = o(g(n))$ implies that there exists some encoding x^* of M such that $g(|x^*|) > c_M f(|x^*|) \cdot \log f(|x^*|)$.

Thus, when we run $D(x^*)$, the simulation has enough time to finish, and so $D(x^*) = 1 - U(x^*, x^*) = 1 - M(x^*)$, a contradiction. □

The same proof goes through for space, but the hierarchy is even tighter because the UTM can simulate space-bounded computation with only a constant-factor overhead.

Theorem 4 (Deterministic Space Hierarchy Theorem)**.** *If f and g are space-constructible and $f(n) = o(g(n))$, then*

$$\mathbf{SPACE}(f(n)) \subsetneq \mathbf{SPACE}(g(n)).$$

2 Nondeterministic Time Hierarchy

A hierarchy theorem also holds for nondeterministic time. What do we have going for us when trying to prove such a theorem?

Good news: It's possible to simulate an arbitrary k-tape NTM by a 3-tape NTM using only a constant factor blowup in runtime. That is, there exists a universal NTM that can simulate any NTM running in time $T(n)$ using time $O(T(n))$.

Bad news: Central to our construction of the language L in the deterministic case was our ability to "flip" the answer produced by the UTM. It isn't so clear how to do this with the output of an NTM because of the asymmetry between its accept and reject conditions. (This is the same issue underlying the $\mathbf{NP}$ vs. $\mathbf{coNP}$ problem.)

Despite this fundamental issue, we can still prove a hierarchy theorem using a more clever diagonalization. Here's the statement:

Theorem 5. *Let f and g be time-constructible functions such that $f(n+1) = o(g(n))$. Then*

$$\mathbf{NTIME}(f(n)) \subsetneq \mathbf{NTIME}(g(n)).$$

The proof in Arora-Barak is due to Zák, but I'll sketch a different (simpler, in my opinion) version due to Fortnow and Santhanam.

As before, we'll define the language in $\mathbf{NTIME}(g(n)) \setminus \mathbf{NTIME}(f(n))$ by an NTM D deciding it. This NTM will expect two inputs x, y. Let ε denote the empty string.

$D(x, y) =$

1. If $|y| < g(|x|)$: Use the UNTM to simulate $M_x(x, y0)$ and $M_x(x, y1)$, stopping each after $g(|x|+|y|)$ time steps have been simulated. Accept iff both runs accept.
2. If $|y| \geq g(|x|)$: Use the UNTM to simulate $M_x(x, \varepsilon)$ using y as the choice of nondeterminism, stopping after $g(|x|)$ steps have been simulated. Flip the answer.

Let L be the language decided by D. The fact that the UNTM runs with constant simulation overhead implies that $L \in \mathbf{NTIME}(g(n))$.

Now suppose for the sake of contradiction that some NTM N decides L in time $\leq cf(n)$ for some constant c. Let x^* be an encoding of N such that $cf(|x^*| + m + 1) \leq g(|x^*| + m)$ for all $m \geq 0$, which exists because $f(n+1) = o(g(n))$. Then

$$(x^*, \varepsilon) \in L \iff N(x^*, 0) \text{ and } N(x^*, 1) \text{ both accept,}$$

since running the machine $D(x^*, \varepsilon)$ enters case 1, and $cf(|x^*| + 1) \leq g(|x^*|)$ implies that both simulations therein have time to run to completion. Since N decides L, this is equivalent to

$$(x^*, y) \in L \; \forall |y| = 1.$$

We can repeat this argument, increasing the length of y one bit at a time, yielding

$$\begin{aligned}(x^*, \varepsilon) \in L &\iff (x^*, y) \in L \; \forall |y| = 1 \\ &\iff (x^*, y) \in L \; \forall |y| = 2 \\ &\dots \\ &\iff (x^*, y) \in L \; \forall |y| = g(|x^*|).\end{aligned}$$

Now for all strings y of length $g(|x^*|)$, running $D(x^*, y)$ enters case 2. So

$$(x^*, y) \in L \; \forall |y| = g(|x^*|) \iff N(x^*, \varepsilon) \text{ rejects for every choice of nondeterminism } |y| = g(|x^*|).$$

Since the runtime of N is bounded by $cf(|x^*|) \leq g(|x^*|)$, this is equivalent to $N(x^*, \varepsilon)$ rejecting overall, which is a contradiction.

3 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 6. *If* $\mathbf{P} = \mathbf{NP}$*, then* $\mathbf{EXP} = \mathbf{NEXP}$.

To prove this, it suffices to show that if $L \in \mathbf{NTIME}(2^{n^c})$ for some constant c, and $\mathbf{P} = \mathbf{NP}$, then $L \in \mathbf{EXP}$. 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:

$$\begin{tikzcd}
L \in \mathrm{NTIME}(2^{n^c}) \arrow[d, "(1)"'] & & L \in \mathrm{EXP} \\
L_{\mathrm{pad}} \in \mathrm{NP} \arrow[rr, Rightarrow, "\mathrm{P} = \mathrm{NP}"] & & L_{\mathrm{pad}} \in \mathrm{P} \arrow[u, "(2)"']
\end{tikzcd}$$

The construction of this padded language is as follows.

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

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

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. □

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. □

4 Ladner's Theorem

[I didn't get to talk about this in class :(, but the theorem is fascinating and the idea behind the proof is really cool, so I encourage you to look at it!]

Ladner's Theorem is an interesting application of padding and diagonalization that 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 7. *If $\mathbf{P} \neq \mathbf{NP}$, there exists a language L such that $L \notin \mathbf{P}$ and L is not $\mathbf{NP}$-complete.*

Here's a brief proof sketch, with more details in Arora-Barak. 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

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

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 $\mathsf{SAT}_T \in \mathbf{P}$. 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 $\mathbf{NP}$-complete $\implies$ $\mathbf{P} = \mathbf{NP}$. So suppose SAT_T is $\mathbf{NP}$-complete. Then $\mathsf{SAT} \leq_p \mathsf{SAT}_T$ 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.