CAS CS 535: Complexity Theory

Lecturer: Mark Bun — Fall 2023

Lecture Notes 4: Finish NP-Completeness, Decision vs. Search, Time Hierarchy

Reading.

- Arora-Barak § 2.4-2.5, 3.1-3.2, 4.1.3

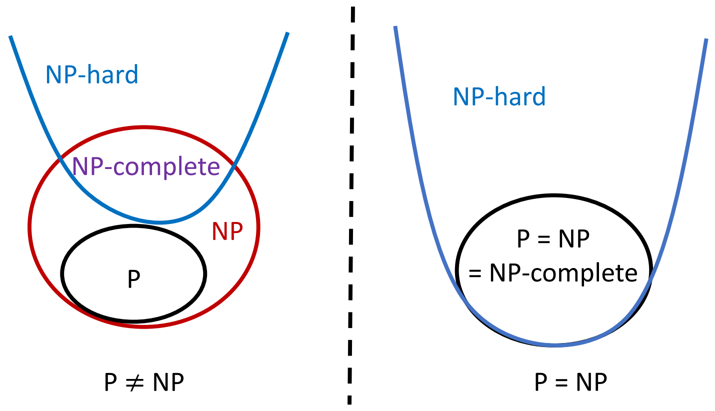
Last time: Reductions, NP-completeness, Cook-Levin Theorem

Definition 1. Language A is poly-time reducible (a.k.a. Karp-reducible, many-to-one reducible) to B if there exists a poly-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that

$$x \in A \iff f(x) \in B.$$

Definition 2. A language B is

- $\mathbf{NP}$-hard if $A \leq_p B$ for every language $A \in \mathbf{NP}$, and
- $\mathbf{NP}$-complete if B is $\mathbf{NP}$-hard <u>and</u> $B \in \mathbf{NP}$.



1 An NP-Completeness Example

In the *hitting set* HS problem, you are given a "ground set" U, natural number k, and sets $S_1, \ldots, S_\ell \subseteq U$. Your goal is to determine whether there exists a set $T \subseteq U$ such that $|T| \leq k$ and $T \cap S_i \neq \emptyset$ for every $i \in [\ell]$.

This is in $\mathbf{NP}$: The certificate is the description of the set T. To verify it, just check that $|T| \leq k$ and that T indeed intersects every set S_i.

To show that it's $\mathbf{NP}$-complete, we can reduce from 3SAT. Let $\varphi = (\ell_1^1 \vee \ell_2^1 \vee \ell_3^1) \wedge \cdots \wedge (\ell_1^m \vee \ell_2^m \vee \ell_3^m)$ be a 3SAT instance on variables $x_1, \ldots, x_n$. Our reduction computes the following HS instance:

$$\begin{aligned} U &= \{x_1, \overline{x_1}, \ldots, x_n, \overline{x_n}\} \\ S_1 &= \{\ell_1^1, \ell_2^1, \ell_3^1\} \\ &\cdots \\ S_m &= \{\ell_1^m, \ell_2^m, \ell_3^m\} \\ S_{m+1} &= \{x_1, \overline{x_1}\} \\ &\cdots \\ S_{m+n} &= \{x_n, \overline{x_n}\} \\ k &= n. \end{aligned}$$

This reduction takes time polynomial in the instance size. To show that it is correct, we need to show that φ is satisfiable $\iff \langle U, S_1, \ldots, S_{m+n}, k\rangle \in \mathsf{HS}$.

$\Rightarrow$: Let $(b_1, \ldots, b_n)$ be a satisfying assignment to φ. We construct a hitting set T as follows. For each $i = 1, \ldots, n$, if $b_i = 1$, place x_i in T. If $b_i = 0$, place $\overline{x_i}$ in T. This gives a hitting set of size $n = k$.

$\Leftarrow$: Let T be a hitting set of size $k = n$. Since T must intersect $S_{m+1}, \ldots, S_{m+n}$, it must contain exactly one element of each pair. Construct the satisfying assignment $(b_1, \ldots, b_n)$ by taking $b_i = 1 \iff x_i \in T$.

2 Decision vs. Search

For much of this class, we focus on decision problems (languages), i.e., those with yes/no answers. Why do we do this when computation encompasses so many other types of problems?

1. This is the simplest setting for studying complexity, where we already can't answer most of the interesting questions we want to ask.
2. Solving decision problems actually lets us solve other types of computational problems!

Today, we'll address the second point by describing about a generic "search-to-decision" reduction for $\mathbf{NP}$. But to set up this general reduction, let's start by looking at a concrete problem.

Recall the decision problem

$$\mathsf{SAT} = \{\varphi \mid \varphi \text{ has a satisfying assignment}\}.$$

This corresponds to a natural search problem:
$\mathsf{SAT} - \mathsf{Search}$: Given a CNF φ, output a satisfying assignment to φ if one exists, and $\perp$ otherwise.

Theorem 3. *If* $\mathsf{SAT} \in \mathbf{P}$*, then there exists a poly-time algorithm solving* $\mathsf{SAT} - \mathsf{Search}$.

Proof. Let A be a poly-time algorithm deciding SAT. The idea is to repeatedly apply A to identify a satisfying assignment bit-by-bit.

First some notation: For a formula φ and $b \in \{0, 1\}$, let $\varphi|_{x_1=b}$ be the formula φ with variable x_1 replaced by the constant b. (And similarly if we want to fix more variables)

Observe that:

$$\varphi \in \mathsf{SAT} \iff \varphi_{x_1=0} \in \mathsf{SAT} \quad \text{or} \quad \varphi_{x_1=1} \in \mathsf{SAT}.$$

The following procedure uses A to construct a satisfying assignment $b_1, \ldots, b_n$ to a formula $\varphi(x_1, \ldots, x_n)$.

```
Alg Find($\varphi$):
    If $A(\varphi) = 0$: Output $\perp$.
    For $i = 1, \ldots, n$:
        If $A(\varphi|_{x_1=b_1,\ldots,x_{i-1}=b_{i-1},x_i=0}) = 1$:        // i.e., $\varphi|_{x_1=b_1,\ldots,x_i=0}(x_{i+1}, \ldots, x_n) \in \mathsf{SAT}$
            Set $b_i = 0$
        Else:
            Set $b_i = 1$
    Output $(b_1, \ldots, b_n)$
```

This algorithm uses $n + 1$ invocations of the poly-time algorithm A, so it runs in poly-time. □

This result is just an example of a more general phenomenon. To state this, we need the following definition of an $\mathbf{NP}$ search problem.

Definition 4. Let $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$ be a poly-time computable function ("verification function"), and let $p : \mathbb{N} \to \mathbb{N}$ be a polynomial ("certificate length"). The $\mathbf{NP}$ search problem associated to f and p is as follows.

Given $x \in \{0,1\}^*$, find a $u \in \{0,1\}^{p(|x|)}$ such that $f(x, u) = 1$ (if one exists).

For example, we can capture the $\mathsf{SAT} - \mathsf{Search}$ problem by taking $f(\varphi, u) = 1 \iff u$ satisfies the CNF φ.

As another example, we can take $f(\langle G, k\rangle, u) = 1 \iff u$ is an independent set for G of size k.

Theorem 5. *If* $\mathbf{P} = \mathbf{NP}$*, then every* $\mathbf{NP}$ *search problem can be solved in poly time.*

Proof. Consider an $\mathbf{NP}$ search problem defined by verification function f and certificate length p. The idea will, again, be to construct a certificate bit-by-bit using our ability to solve a decision problem associated to f in poly-time.

One choice of decision problem that works here is

$$L_f = \{\langle x, b\rangle \mid \exists z \in \{0,1\}^{p(|x|)-|b|} \text{ s.t. } f(x, b \circ z) = 1\}.$$

That is, L_f consists of pairs (x, b) such that there exists a way to extend the prefix b to a certificate $u = b \circ z$ for which $f(x, u) = 1$.

The language $L_f \in \mathbf{NP}$ because f is poly-time computable. Therefore, if $\mathbf{P} = \mathbf{NP}$, there is a poly-time algorithm A deciding L_f. This gives us the following search-to-decision reduction:

```
Alg Find($x$):
    If $A(x) = 0$: Output $\perp$.
    For $i = 1, \ldots, p(|x|)$:
        If $A(x, b_1, \ldots, b_{i-1}, 0) = 1$:        // i.e., $\langle x, b_1, \ldots, b_{i-1}, 0\rangle \in L_f$
            Set $b_i = 0$
        Else:
            Set $b_i = 1$
    Output $(b_1, \ldots, b_{p(|x|)})$.
```

□

3 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 6. 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 7 (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, as a consequence, shows that 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	$\cancel{0}$ 1	1	0	0	...
x_2	1	$\cancel{1}$ 0	0	000	
x_3	11	10	$\cancel{1}$ 1	0	
x_4	1	0	1	$10\cancel{0}1$ 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,

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

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 $g(|x|)$ time steps.
2. If U has output 1, return 0. Otherwise, return 1.

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

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 matters. Let us assume that every TM has infinitely many encodings (say, by encoding 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. $\square$

Next time: Nondeterministic time hierarchy, Ladner's theorem, limits of diagonalization.