Notes on Complexity Theory Last updated: September 14, 2011

Lecture 5

Jonathan Katz

1 Diagonalization, Continued

1.1 Ladner's Theorem

We know that there exist $\mathcal{NP}$-complete languages. Assuming $\mathcal{P} \neq \mathcal{NP}$, any $\mathcal{NP}$-complete language lies in $\mathcal{NP} \setminus \mathcal{P}$. Are there languages that are neither in $\mathcal{P}$ nor $\mathcal{NP}$-complete? Ladner's theorem tells us that there are.

As some intuition for Ladner's theorem, take some language $L \in \mathcal{NP} \setminus \mathcal{P}$. Using padding, we will make L "easy enough" so that it can't be $\mathcal{NP}$-complete, while keeping it "hard enough" so it is not in $\mathcal{P}$ either. Say the best algorithm for deciding L runs in time $n^{\log n}$ for concreteness. (The same argument, though messier, works as long as the best algorithm deciding L requires super-polynomial time.) Define

$$L' = \{(x, y) \mid x \in L \text{ and } |x| + |y| = |x|^{\log\log|x|}\}.$$

If $L' \in \mathcal{P}$, then L would be decidable in time $n^{O(\log\log n)}$, a contradiction. On the other hand, L' *is* decidable in time $N^{\log N}$ where N is such that $N^{\log\log N} = n$ (the input length). We have $N = n^{o(1)}$, and so L' is decidable in time $n^{o(\log\log(n^{o(1)}))}$. If L were Karp-reducible to L', then L would be solvable in time $n^{o(\log n)}$, a contradiction. The main challenge in making the above formal is that it is hard to pin down the "best" algorithm for deciding some language L, or that algorithm's exact running time.

Theorem 1 *Assuming $\mathcal{P} \neq \mathcal{NP}$, there exists a language $A \in \mathcal{NP} \setminus \mathcal{P}$ which is not $\mathcal{NP}$-complete.*

Note: We did not cover the proof of Ladner's theorem in class, but one is included here for completeness.

Proof The high-level intuition behind the proof is that we construct A by taking an $\mathcal{NP}$-complete language and "blowing holes" in it in such a way that the language is no longer $\mathcal{NP}$-complete yet not in $\mathcal{P}$ either. The specific details are quite involved.

Let $M_1, \ldots$ denote an enumeration of all polynomial-time Turing machines with boolean output; formally, this can be achieved by considering an enumeration[1] of $\mathcal{M} \times \mathbb{Z}$ (where $\mathcal{M}$ is the set of Turing machines), and defining M_i as follows: if the i^{th} item in this enumeration is (M, j), then $M_i(x)$ runs $M(x)$ for at most $|x|^j$ steps. We remark that $M_1, \ldots$ also gives an enumeration of languages in $\mathcal{P}$ (with languages appearing multiple times). In a similar way, let $F_1, \ldots$ denote an enumeration of polynomial-time Turing machines without the restriction of their output length. Note that this gives an enumeration of functions computable in polynomial time.

Define language A as follows:

$$A = \{x \mid x \in \mathsf{SAT} \wedge f(|x|) \text{ is even}\},$$

[1] Since both $\mathcal{M}$ and $\mathbb{Z}$ are countable, it follows that $\mathcal{M} \times \mathbb{Z}$ is countable.

for some function f that remains to be defined. Note that as long as we ensure that f is computable in polynomial time, then $A \in \mathcal{NP}$. We define f by a polynomial-time Turing machine M_f that computes it. Let M_{SAT} be a machine that decides SAT (not in polynomial time, of course...), and let $f(0) = f(1) = 2$. On input 1^n (with $n > 1$), M_f proceeds in two stages, each lasting for exactly n steps:

1. During the first stage, M_f computes $f(0), f(1), \ldots$ until it runs out of time. Suppose the last value of f it was able to compute was $f(x) = k$. The output of M_f will be either k or $k+1$, to be determined by the next stage.

2. Then:

   - If $k = 2i$ is even, then M_f tries to find a $z \in \{0,1\}^*$ such that $M_i(z)$ outputs the "wrong" answer as to whether $z \in A$. (That is, M_f tries to find z such that either $z \in A$ but $M_i(z) = 0$, or the opposite.) This is done by computing $M_i(z), M_{\text{SAT}}(z)$, and $f(|z|)$ for all strings z in lexicographic order. If such a string is found within the allotted time, the output of M_f is $k+1$. Otherwise, the output of M_f is k.
   - If $k = 2i-1$ is odd, then M_f tries to find a string z such that $F_i(z)$ is an incorrect Karp reduction from SAT to A. (That is, M_f tries to find a z such that either $z \in \mathsf{SAT}$ but $F_i(z) \notin A$, or the opposite.) This is done by computing $F_i(z)$, $M_{\text{SAT}}(z)$, $M_{\text{SAT}}(F_i(z))$, and $f(|F_i(z)|)$. If such a string is found within the allotted time, then the output of M_f is $k+1$; otherwise, the output is k.

By its definition, M_f runs in polynomial time. Note also that $f(n+1) \geq f(n)$ for all n.

We claim that $A \notin \mathcal{P}$. Suppose the contrary. Then A is decided by some M_i. In this case, however, the second stage of M_f with $k = 2i$ will never find a z satisfying the desired property, and so f is eventually a constant function and in particular $f(n)$ is odd for only finitely-many n. But this implies that A and SAT coincide except for finitely-many strings. But this implies that $\mathsf{SAT} \in \mathcal{P}$, a contradiction to our assumption that $\mathcal{P} \neq \mathcal{NP}$.

Similarly, we claim that A is not $\mathcal{NP}$-complete. Suppose the contrary. Then there is a polynomial-time function F_i which gives a Karp reduction from SAT to A. Now $f(n)$ will be even for only finitely-many n, implying that A is a finite language. But then $A \in \mathcal{P}$, a contradiction to our assumption that $\mathcal{P} \neq \mathcal{NP}$. ■

As an addendum, we note that (assuming $\mathcal{P} \neq \mathcal{NP}$, of course) we know of no "natural" languages provably in $\mathcal{NP} \setminus \mathcal{P}$ that are not $\mathcal{NP}$-complete. However, there are a number of languages conjectured to fall in this category, including graph isomorphism and essentially all languages derived from cryptographic assumptions (e.g., factoring).

1.2 Relativizing the $\mathcal{P}$ vs. $\mathcal{NP}$ Question

We conclude by showing some limitations of the diagonalization technique. (Interestingly, these limitations are proven by diagonalization!). Informally, diagonalization relies on the following properties of Turing machines:

1. The fact that Turing machines can be represented by finite strings.

2. The fact that one Turing machine can simulate another (without much overhead).

Any proof that relies only on these facts is essentially treating Turing machines as black boxes (namely, looking only at their input/output), without caring much about the details of how they work. In that case, the proof should apply just as well to *oracle* Turing machines.

An oracle is just a function $\mathcal{O} : \{0,1\}^* \to \{0,1\}$, and of course for any $\mathcal{O}$ we have a corresponding language L. Fixing $\mathcal{O}$, an oracle Turing machine $M^{\mathcal{O}}$ is given the ability to make "queries" to $\mathcal{O}$ and obtain the result in a single time step.[2] (We have already seen this notation when we talked about Cook-Turing reductions.) Fixing some $\mathcal{O}$, we say $L \in \mathcal{P}^{\mathcal{O}}$ if there exists a polynomial-time Turing machine M such that $x \in L \Leftrightarrow M^{\mathcal{O}}(x) = 1$. Similarly, $L \in \mathcal{NP}^{\mathcal{O}}$ if there exists a polynomial-time Turing machine M such that $x \in L \Leftrightarrow \exists w : M^{\mathcal{O}}(x, w) = 1$. More generally, for any class $\mathcal{C}$ defined in terms of Turing machines deciding languages in that class, we can define the class $\mathcal{C}^{\mathcal{O}}$ in the natural way.

Given a result about two complexity classes $\mathcal{C}_1, \mathcal{C}_2$, we can ask whether that same result holds about $\mathcal{C}_1^{\mathcal{O}}, \mathcal{C}_2^{\mathcal{O}}$ for *any* oracles $\mathcal{O}$. If so, then the result *relativizes*. Any result proved via diagonalization, as defined above, relativizes. As examples: the result about universal simulation relativizes, as does the time-hierarchy theorem.

We now show that the $\mathcal{P}$ vs. $\mathcal{NP}$ question does *not* relativize. We demonstrate this by showing that there exists oracles A, B such that

$$\mathcal{P}^A = \mathcal{NP}^A \text{ but } \mathcal{P}^B \neq \mathcal{NP}^B.$$

When this result was first demonstrated [3], it was taken as an indication of the difficulty of resolving the $\mathcal{P}$ vs. $\mathcal{NP}$ question using "standard techniques". It is important to note, however, that various non-relativizing results are known. As one important example, the proof that SAT is $\mathcal{NP}$-complete does not relativize. (This is not the best example, since SAT is a problem and not a class.) See [5, Lect. 26] and [2, 4, 6] for further discussion.

An oracle A for which $\mathcal{P}^A = \mathcal{NP}^A$. Recall that $\mathsf{EXP} = \bigcup_c \text{TIME}(2^{n^c})$. Let A be an EXP-complete language. It is obvious that $\mathcal{P}^A \subseteq \mathcal{NP}^A$ for any A, so it remains to show that $\mathcal{NP}^A \subseteq \mathcal{P}^A$. We do this by showing that

$$\mathcal{NP}^A \subseteq \mathsf{EXP} \subseteq \mathcal{P}^A.$$

The second inclusion is immediate (just use a Karp reduction from any language $L \in \mathsf{EXP}$ to the EXP-complete problem A), and so we have only to prove the first inclusion. This, too, is easy: Let $L \in \mathcal{NP}^A$ and let M be a polynomial-time non-deterministic machine such that M^A decides L. Then using a deterministic exponential-time machine M' we simply try all possible non-deterministic choices for M, and whenever M makes a query to A we have M' answer the query by itself.

An oracle B for which $\mathcal{P}^B \neq \mathcal{NP}^B$. This is a bit more interesting. We want to find an oracle B such that $\mathcal{NP}^B \setminus \mathcal{P}^B$ is not empty. For any oracle (i.e., language) B, define language L_B as follows:

$$L_B \stackrel{\text{def}}{=} \{1^n \mid B \cap \{0,1\}^n \neq \emptyset\}.$$

It is immediate that $L_B \in \mathcal{NP}^B$ for any B. (On input 1^n, guess $x \in \{0,1\}^n$ and submit it to the oracle; output 1 iff the oracle returns 1.) As a "warm-up" to the desired result, we show:

[2] There are subtleties in dealing with space-bounded oracle machines. We only discuss time-bounded oracle machines here.

Claim 2 *For any deterministic, polynomial-time oracle machine M, there exists a language B such that M^B does not decide L_B.*

Proof Given M with polynomial running time $p(\cdot)$, we construct B as follows: let n be the smallest integer such that $2^n > p(n)$. Note that on input 1^n, machine M cannot query its oracle on all strings of length n. We exploit this by defining B in the following way:

> Run $M(1^n)$ and answer "0" to all queries of M. Let b be the output of M, and let $Q = \{q_1, \ldots\}$ denote all the queries of length exactly n that were made by M. Take arbitrary $x \in \{0,1\}^n \setminus Q$ (we know such an x exists, as discussed above). If $b = 0$, then put x in B; if $b = 1$, then take B to just be the empty set.

Now $M^B(1^n) = b$ (since B returns 0 for every query made by $M(1^n)$), but this answer is incorrect by construction of B. ■

This claim is not enough to prove the desired result, since we need to reverse the order of quantifiers and show that there exists a language B such that for *all* deterministic, polynomial-time M we have that M^B does not decide L_B. We do this by extending the above argument. Consider an enumeration $M_1, \ldots$ of all deterministic, polynomial-time machines with running times $p_1, \ldots$. We will build B inductively. Let $B_0 = \emptyset$ and $n_0 = 1$. Then in the i^{th} iteration do the following:

- Let n_i be the smallest integer such that $2^{n_i} > p_i(n_i)$ and also $n_i > p_j(n_j)$ for all $1 \leq j < i$.
- Run $M_i(1^{n_i})$ and respond to its queries according to B_{i-1}. Let $Q = \{q_1, \ldots\}$ be the queries of length exactly n_i that were made by M_i, and let $x \in \{0,1\}^{n_i} \setminus Q$ (again, we know such an x exists). If $b = 0$ then set $B_i = B_{i-1} \cup \{x\}$; if $b = 1$ then set $B_i = B_{i-1}$ (and so B_i does not contain any strings of length n_i).

Let $B = \bigcup_i B_i$. We claim that B has the desired properties. Indeed, when we run $M_i(1^{n_i})$ with oracle access to B_i, we can see (following the reasoning in the previous proof) that M_i will output the wrong answer (and thus $M_i^{B_i}$ does not decide L_{B_i}). But the output of $M_i(1^{n_i})$ with oracle access to B is the same as the output of $M_i(1^{n_i})$ with oracle access to B_i, since all strings in $B \setminus B_i$ have length greater than $p_i(n_i)$ and so none of M_i's queries (on input 1^{n_i}) will be affected by using B instead of B_i. It follows that M_i^B does not decide L_B.

2 Space Complexity

Recall that for space complexity (in both the deterministic and non-deterministic cases) we measure the number of cells used on the *work tape* only. This allows us to talk meaningfully of sublinear-space algorithms, and algorithms whose output may be longer than the space used.

Note also that in the context of space complexity we may assume without loss of generality that machines have only a single work tape. This is so because we can perform universal simulation of a k-tape Turing machine on a Turing machine with just a single work tape, with only a constant factor blowup in the space complexity.

When we talk about non-deterministic space complexity we refer to our original notion of non-deterministic Turing machines, where there are two transition functions and at every step the machine makes a non-deterministic choice as to which one to apply. It turns out that, just as we did in the case of $\mathcal{NP}$, we can give a "certificate-based" definition of non-deterministic space

classes as well, though we need to be a little careful since the length of the certificate may exceed the space-bound of the machine. In particular, we imagine a (deterministic) machine with an input tape and work tape as usual, and also a special *certificate tape*. When measuring the space used by this machine, we continue to look at the space on the work tape only. The certificate tape (like the input tape) is a *read-only* tape; moreover (and unlike the input tape), we restrict the Turing machine so that it may only move its certificate-tape head from left to right (or stay in place). This gives a definition equivalent to the definition in terms of non-deterministic Turing machines; in particular:

Claim 3 *$L \in \text{NSPACE}(s(n))$ iff there exists a (deterministic) Turing machine with a special "read-once" certificate tape as described above that uses space $O(s(n))$ (where n is the input length, and is independent of the certificate length), and such that $x \in L$ iff there exists a certificate w such that $M(x, w) = 1$.*

If the certificate-tape head is allowed to move back-and-forth across its tape, this gives the machine significantly more power; in fact, if we consider log-space machines that move freely on their certificate tape we get the class $\mathcal{NP}$! See [5, Chap. 5] for further discussion regarding the above.

Bibliographic Notes

The intuition before the proof of Ladner's theorem is due to Russell Impagliazzo (personal communication).

References

[1] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[2] E. Allender. Oracles versus Proof Techniques that Do Not Relativize. *SIGAL Intl. Symposium on Algorithms*, pp. 39–52, 1990.

[3] T. Baker, J. Gill, and R. Solovay. Relativizations of the $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ Question. *SIAM J. Computing* 4(4): 431–442, 1975.

[4] L. Fortnow. The Role of Relativization in Complexity Theory. *Bulletin of the European Association for Theoretical Computer Science*, 52: 229–243, 1994.

[5] O. Goldreich. Introduction to Complexity Theory (July 31, 1999).

[6] J. Hartmanis, R. Chang, S. Chari, D. Ranjan, and P. Rohatgi. Relativization: A Revisionist Retrospective. *Current Trends in Theoretical Computer Science*, 1993. Available from `http://www.cs.umbc.edu/ chang/papers/revisionist`.