18.405J/6.841J Advanced Complexity Theory February 12, 2007

Lecture 2

Lecturer: Madhu Sudan *Scribe: Alan Deckelbaum*

1 Administrivia

- Make sure you're on the mailing list. You should have received an email earlier today.
- Sign up for scribing.
- Swastik's office hours:
  - Thursdays 6-8pm Gates 5^{th} floor
  - Tuesday 2/20 (same location)

2 Overview

The topic of lecture today is diagonalization. We will cover the following main points:

- $NTIME(o(n^2)) \subsetneq NTIME(n^{10})$
- Ladner's Theorem
- Relativization

3 Introduction to Diagonalization

Diagonalization is the main technique we have for proving lower bounds. (However, it is very unlikely that we will be able to prove $P \neq NP$ using diagonalization, as we will see at the end of the lecture.) Our first example is to use diagonalization to prove that $TIME(o(n^2)) \subsetneq TIME(n^{10})$. The idea is to enumerate all deterministic Turing Machines that run in $TIME(n^2)$. Call these machines

$$M_1, M_2, M_3, \ldots, M_i, \ldots$$

and let L_i be the language accepted by M_i. Also, we will enumerate all binary strings

$$x_1, x_2, x_3, \ldots, x_i, \ldots$$

We will define $L_j(x_i) = 1$ if $x_i \in L_j$, 0 otherwise. We can now imagine a large binary matrix with the M's going across and the x's going down, so that a_{ij} in the matrix is equal to $L_j(x_i)$. We construct the language L to be the diagonal complement of the matrix. In other words, $x_i \in L$ if and only if $x_i \notin L_i$. Thus, we see that $L \neq L_i$ for any i, and therefore $L \notin TIME(n^2)$. It remains to show that $L \in TIME(n^{10})$. In other words, we must construct a machine M running in n^{10} time such that $L(M) = L$. We will need two basic primitives in order to create this machine:

- **Simulation** - M should be able to simulate M_i for every i. By careful simulation, this can be done without overstepping the time bounds. (Even if the simulation is sloppy, it is not very difficult to simulate M_i with only a quadratic increase in time required.)
- **Complementation** - M needs to be able to complement the result of M_i's computation.

We would also like a way to enumerate all of the M_i's. Notice that it is possible to construct a language that differs from each M_i on infinitely many inputs. For example, we can use the sequence

$$M_1, M_1, M_2, M_1, M_2, M_3, M_1, M_2, M_3, M_4 \ldots$$

This enumeration also avoids potential difficulties with constant factors on running time: M will eventually differ with M_i on some input that is large enough that it can be computed in the running time of our machine.

All of these techniques for diagonalization depend on our ability to take the complement of a machine's computation. When the complexity class isn't closed under complementation, this can be nontrivial. For example, the question arises of how one might be able to use a similar idea to show that a language isn't in NP.

4 $NTIME(n^2) \subsetneq NTIME(n^{10})$

For now, we will focus on a single $NTIME(n^2)$ machine, M, of length i. We want to construct a language $L \neq L(M)$ (for some sufficiently long strings of length $\geq i$). Furthermore, we want to be able to prove that $L \in NTIME(n^{10})$. We notice immediately that $NTIME(n^2) \subsetneq NTIME(2^{n^3})$, as $NTIME(n^2) \subseteq TIME(2^{n^2}) \subsetneq TIME(2^{n^3}) \subseteq NTIME(2^{n^3})$.

We could come up with a sequence of increasingly growing functions, say

$$T_1(n), T_2(n), \ldots, T_k(n)$$

where $T_1(n) = n^2$, and $T_k(n) = 2^{n^3}$. We then know that there must be some i such that

$$NTIME(T_i(n)) \neq NTIME(T_{i+1}(n))$$

since $NTIME(T_1(n)) \neq NTIME(T_k(n))$.

Take the machine M, and look at all inputs between $0^i, 0^{i+1}, 0^{i+2}, \ldots, 0^I$, for some very large I. (The size of I will be specified in more detail later.) We will construct the language L, which shifts the above language by one 0 of the input. In other words, 0^k is in L if and only if 0^{k+1} is in $L(M)$, for all k between i and $I-1$. We have $L \neq L(M)$ as long as it is not the case that either $0^k \in L(M)$ for all $k \in \{i, i+1, \ldots, I\}$, or instead $0^k \notin L(M)$ for all $k \in \{i, i+1, \ldots, I\}$. Our idea to solve this potential difficulty is to *deterministicially* simulate M on 0^i, and set O^I to be the complement of this result. If I is large enough, we will be able to deterministically simulate M on 0^i without overstepping the time bounds. Provided that $I >> i$, we would have

$$L(M) \in NTIME(T(n)) \Rightarrow L \in NTIME(T(n+1))$$

All that remains is to make I large enough. This can be done by ensuring

$$2^{T(i)} < T(I)$$

which can be accomplished provided that we can compute T easily. In fact, this approach can be used to show that $NTIME(n^2) \neq NTIME(o(n^2))$.

The proof combining the above approach with enumeration was not shown in lecture, but an outline appears below. (The theorem was proved by Cook, and this particular proof is from Fortnow's survey on diagonalization. See van Melkebeek's paper for more information.)

Let $M_1, M_2, \ldots$ be an enumeration of $NTIME(T(n))$ machines.

I first define the following subprocedure, which calculates j (the index of the machine to be diagonalized against), as well as i and I (where i and I are defined as above).

COMPUTE-INDICES: On input O^n:

1. Set $j \leftarrow 1, i \leftarrow 1, I \leftarrow \infty$
2. While $I < n$:
   (a) Let I be the smallest integer such that $NTIME(T(i)) \subset DTIME(T(I))$.
   (b) If $I \geq n$, break.
   (c) Set $j \leftarrow j + 1, i \leftarrow I + 1$
3. Return (j, i, I).

We define the language L such that $L(0^n)$ is computed by the following procedure:

On input O^n:

1. Run COMPUTE-INDICES(0^n) to determine j, i, and I.
2. If $n = I$ then set $L(0^I) =$ the opposite of $M_j(0^i)$
3. Otherwise, set $L(O^i) = M(0^{i+1})$

We see that COMPUTE-INDICES simply returns the smallest index j such that M_j hasn't already been diagonalized against. (If $I < n$, then we can assume that on some smaller input, L already differs from the machine whos index is j.) COMPUTE-INDICES thus returns the smallest j such that the corresponding I is $\geq n$, and thus M_j hasn't already been diagonalized against. The actual procedure for L then runs the algorithm described above for diagonalizing against a single NTM. With an efficient method of simulating another NTM, this yields the following theorem:

Theorem 1 (Cook) *For every time-constructible function $T(n)$,*

$$NTIME(o(T(n))) \subsetneq NTIME(T(n+1))$$

This proof uses lazy diagonalization and a very effective simulation of a $NTIME(o(T(n)))$ machine.

5 Ladner's Theorem

5.1 Background

NP-completeness was introduced in the 1970's.

- Cook '70 $\leftarrow$ defined NP-completeness
- Karp '72 $\leftarrow$ gave many examples of NP-complete problems
- Levin '72 $\leftarrow$ independently did work similar to that of both Cook and Karp.

Levin's advisor, Kolmogorov, asked Levin about graph isomorphism, factoring, and linear programming, three problems that we didn't know whether they were in P and also didn't have a proof of their NP-completeness. (We now know that linear programming $\in P$.)

We ask the question: for all $L \in NP$, is it true that either $L \in P$ or L is NP-complete?

5.2 Ladner's Theorem

Ladner showed that if $NP \neq P$, then the answer to the above question is negative. Diagonalization is once again the tool to prove this.

Assume $NP \neq P$. We want a language $L \in NP$ such that L isn't NP-complete and $L \notin P$. We've seen examples of proving that something isn't in P, but we haven't yet dealt with a situation of showing that something isn't NP-complete. Our proof will be based on the fact that if L is NP-complete, then SAT can be decided by a deterministic polynomial time machine with an oracle for L.

Notation: M^L is an algorithm M using an "oracle" for language L as a subroutine. $M^L \in P^L$ implies that M runs in polynomial time, assuming it takes unit time to decide L.

Let

$$M_1, M_2, \ldots, M_i, \ldots$$

be an enumeration of polynomial time oracle machines. We would like SAT $\neq \mathrm{M}_1^{\mathrm{L}}, \mathrm{M}_2^{\mathrm{L}}, \ldots$ (as L is not NP-complete). Furthermore, as $L \notin P$, we would like $L \neq M_1^{\emptyset}, M_2^{\emptyset}, \ldots$. (Recall that $\emptyset$ means an oracle for the empty language, which can clearly be simulated in polynomial time.)

Our goal is to construct a language that looks like SAT some of the time and like $\emptyset$ the rest of the time. Look at the sequence

$$M_1^L, M_1^{\emptyset}, M_2^L, M_2^{\emptyset}, M_3^L, M_3^{\emptyset}, \ldots$$

We want to construct L such that L is not $M_i^{\emptyset}$ (after a finite prefix), and that SAT is not M_i^L (after a finite prefix) for any i. We define L as follows:

On input x, the machine N deciding L will go sequentially go through the above enumeration of machines to ensure that machines with L-oracles differ from SAT, and machines with $\emptyset$-oracles differ from L. (If a machine queries the L oracle, N is able to simulate itself. Notice that N might run out of computation time when performing this simulation. This case will be dealt with shortly.)

I begin by defining the following subprocedure:

DISAGREE(Polynomial Time Oracle Machine M, string y):

1. If M is a machine with an L-oracle:
   (a) Simulate M on y, and test if $SAT(y)$. If these simulations return opposite results, return YES. Otherwise, return NO.
2. If instead M is a machine with a $\emptyset$-oracle:
   (a) Simulate M on y, and test if $L(y)$. If these simulations return opposite results, return YES. Otherwise, return NO.

Now, on a given input x, we can define N's behavior as follows:

On input x:

1. Set $i \leftarrow 1$
2. Set string $y =$ "0"
3. Until N exceeds its polynomial time bound:
   (a) While $DISAGREE(M_i^{\emptyset}, y)$ returns NO, increment y in lexicographic order

(b) While $DISAGREE(M_i^L, y)$ returns NO, increment y in lexicographic order

(c) Set $i \leftarrow i + 1$

4. When N is about to exceed its time bound:

   - If N was in the process of running DISAGREE with some machine $M_i^\emptyset$, return $SAT(x)$. If instead N was in the process of running DISAGREE with some machine M_i^L, return 0.

First, I will show that L is not NP-complete. If L were NP complete, then some M_i^L would be computing SAT. In this case, $DISAGREE(M_i^L, y)$ would always return NO, and thus N would get stuck on this loop in step 3b. However, in this case, L would be equal to all 0's after a finite prefix, and thus $L \in P$. However, the combination of L being NP-complete and $L \in P$ implies that $P = NP$, contradicting our assumption.

It is clear, on the other hand, that $L \in NP$. A nondeterministic polynomial time machine (with knowledge of N's time bound) can simply simulate N on a input x. We can then manually solve $SAT(x)$ if necessary, since $SAT \in NP$.

It remains now to show that $L \notin P$. If $L \in P$, then it must be the case that for all sufficiently large inputs, L is the same language as $M_i^\emptyset$ for some i. In this case, $DISAGREE(M_i^\emptyset, y)$ returns NO for all sufficiently large y, and thus L is the same as SAT after some finite prefix. However, if $L = SAT$ after a finite prefix and $L = M_i^\emptyset$, then we would have $SAT \in P$, which contradicts our initial assumption that $P \neq NP$.

6 Diagonalization and the P vs. NP Question

A paper by Baker, Gill, and Solovay asked whether diagonalization could be used to prove $P \neq NP$. The "politically correct" answer is simply that this depends on the truth of $P \neq NP$. The functional answer provided by Baker, Gill, and Soloway is based on the fact that diagonalization proofs relativize. However, the proof of $P \neq NP$ would not relativize.

Enumerate $M_1, M_2, \ldots$ of oracle polynomial time machines. If each machine is given an oracle for some language O, we get a collection $\{M_1^O, M_2^O, \ldots\}$, the set of languages decided in P^O. Furthermore, we have $\{M_1^\emptyset, M_2^\emptyset, \ldots\} = P$. Similarly, by enumerating NTM's with access to oracles, we can obtain the class NP^O, etc.

Diagonalization doesn't distinguish between the particular oracle used. Thus, if diagonalization shows $P \neq NP$, it means that $P^O \neq NP^O$ for all oracles O. Therefore, the following two results demonstrate that it is not likely that diagonalization can be used to show $P \neq NP$.

Theorem 2 *There exists an oracle O such that $P^O = NP^O$.*

Proof Let O be a PSPACE-complete language, such as TQBF. We then have $P^{TQBF} = PSPACE = NPSPACE = NP^{TQBF}$. ■

Theorem 3 *There exists an oracle O such that $P^O \neq NP^O$.*

The proof uses diagonalization. A language that demonstrates this result is

$$L^O = \{x | \exists y \text{ such that } |\mathrm{y}| = |\mathrm{x}| \text{ and } \mathrm{O(y)} = 1\}$$

We see that $L^O \in NP^O$ for all O. We want to choose an O such that $L^O \notin P^O$. We will ensure that $L_O \neq M_j^O$ for inputs of length greater than or equal to i. Look at all of the queries that M_j makes to O when it computes on input x, where x of a size larger than any strings that have been previously decided whether or not they are in the language thus far. For all inputs of length less than x, O answers whatever it has previously returned. The oracle also sets $O(x) = 0$, and returns 0 for all strings of length greater than

or equal to $|x|$ that are queried by the machine. As M_j can only ask polynomially many queries to O, for sufficiently large i the polynomial in i will be less than 2^i, and thus there are other strings of length i that M_j has not queried. The value of these inputs can be set to negate the answer to $M_j^O(0^i)$. On the other hand, $L^O \in NP$, since the NP machine can nondeterministically query O on all strings of the appropriate length, and the machine accepts if any one of these queries accepts.