

## 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<sup>th</sup> 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

$$M_1, M_2, M_3, \dots, M_i, \dots$$

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

$$x_1, x_2, x_3, \dots, x_i, \dots$$

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

$$M_1, M_1, M_2, M_1, M_2, M_3, \dots$$

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

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

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  $0^i, 0^{i+1}, 0^{i+2}, \dots, 0^I$ , 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  $k \in \{i, i+1, \dots, I\}$ , or instead  $0^k \notin L(M)$  for all  $k \in \{i, i+1, \dots, I\}$ . Our idea to solve this potential difficulty is to *deterministically* simulate  $M$  on  $0^i$ , and set  $0^I$  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  $I \gg i$ , 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 complete proof combining the above approach with enumeration was not shown in lecture, but appears in the lecture notes. (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.)

**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 ← defined NP-completeness
- Karp '72 ← gave many examples of NP-complete problems
- Levin '72 ← 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

$$M_1, M_2, \dots, M_i, \dots$$

be an enumeration of polynomial time oracle machines. We would like  $SAT \neq M_1^L, M_2^L, \dots$  (as  $L$  is not NP-complete). Furthermore, as  $L \notin P$ , we would like  $L \neq M_1^\emptyset, M_2^\emptyset, \dots$  (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

$$M_1^L, M_1^\emptyset, M_2^L, M_2^\emptyset, M_3^L, M_3^\emptyset, \dots$$

We want to construct  $L$  such that  $L$  is not any of these languages. On input  $x$ , the machine deciding  $L$  will go sequentially go through the above languages to ensure that it differs from each language on some input. Specifically, on a given input, it will begin comparing the results of its own simulation on the first string in lexicographic order to the result of the first language in the above sequence on the same string. If the languages differ, the machine deciding  $L$  will then proceed to test the next string against the next language in the list. If the languages do not differ, the machine will test the same language against the next string in lexicographic order. The machine additionally keeps track of how much time it has used. When the machine runs out of time, it does the following, depending on which language it is currently comparing itself against:

For some  $j$ ,  $L$  may be comparing itself against  $M_j^\emptyset$ . If this is the case, we will set  $L(x) = SAT(x)$ .

For some  $j$ ,  $L$  may be comparing itself against  $M_j^L$ . If this is the case, we will set  $L(x) = 0$ .

All that remains is 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 as  $SAT$ . (Otherwise,  $L$  would disagree with each of the  $M_j^\emptyset$  on some input and it wouldn't be in  $P$ .) However, if  $L = SAT$  and  $L = M_j^\emptyset$ , then we would have  $SAT \in P$ , which contradicts our initial assumption that  $P \neq NP$ .

**Exercise:** Prove that there is an undecidable problem that is not NP-hard.

## 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 Solovay is based on the fact that diagonalization proofs relativize. However, the proof of  $P \neq NP$  would not relativize.

Enumerate  $M_1, M_2, \dots$  of oracle polynomial time machines. If each machine is given an oracle for some language  $O$ , we get a collection  $\{M_1^O, M_2^O, \dots\}$ , the set of languages decided in  $P^O$ . Furthermore, we have  $\{M_1^\emptyset, M_2^\emptyset, \dots\} = P$ . 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

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

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 declared to be in the language thus far. For all inputs of length less than  $x$ ,  $O$  answers whatever it has previously returned. The oracle also sets  $O(x) = 0$ . As  $M_j$  can only ask polynomial 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  $M_j^O(0^i)$ .