6.841 Advanced Complexity Theory February 9, 2009

Lecture 2

Lecturer: Madhu Sudan *Scribe: Sam McVeety*

Today, we will study diagonalization and Ladner's Theorem, which states roughly that:

$$\text{P} \neq \text{NP} \Rightarrow \exists \text{ NP-intermediate Problem}$$

Later, we will look at relativization, in the context of why diagonalization can't prove P $\neq$ NP.

1 Diagonalization

All subsequent applications of diagonalization are based on the method discovered by Cantor (1880s), which he used to prove the uncountability of rational numbers. We outline that method below:

- We can enumerate the rational numbers, r_i, and use them as rows in a matrix
- Examine the binary expansion of each number, one digit in each column
- Construct a new number n such that n does not equal r_i at the ith place, negating along the "diagonal"

Recall that we first applied this to languages to show that they are not computable, and then moved to prove further results. The proofs are very similar, as we can simply change the rows from rational numbers to languages, and the column headings to be an element in the set of all finite strings. For entries in the table, 1 implies membership in the language, 0 implies otherwise.

Proving the uncomputability of the Halting Problem represents a significant step forward (1940s). The jump from "a language is not computable" to "this language is not computable" is nontrivial, particularly given that we have a concise description of the language.

Taking the technique still further, we can use diagonalization to prove results for time and space complexity. Rabin and Blum contributed significantly to these results.

Theorem 1 (Hartmanis, Stearns, '65)

$$\text{TIME}(f(n)) \subsetneq \text{TIME}(\omega(f(n)) \log f(n))$$

$$\text{SPACE}(f(n)) \subsetneq \text{SPACE}(\omega(f(n)))$$

Sketch of Proof

- Describe language L (or Turing Machine M) such that $M \in \text{TIME}(\omega(f(n)) \log f(n))$
- For every $L_i \in \text{TIME}(f(n))$, $L_i \neq L$
- We ensure this inequality by using different "phases" for different i
- On input x, simulate $L_i(x)$ and negate its answer. This is the language L.

■

Note that negation is easy for deterministic machines, but it is more subtle for nondeterministic machines.

Now, we will look at Ladner's Theorem. First, a bit of context for this particular result. In his paper on NP-completeness, Levin managed to prove many things to be NP-complete, with the major exceptions of LP, GRAPHISO, and Factoring. Because so many problems were NP-complete, theorists began to wonder whether problems were either NP-complete or in P. (Concisely, is P$\cup$NP-complete = NP?) Ladner's Theorem answers this question in the negative (unless P = NP).

Theorem 2 (Ladner's Theorem) *If* $\mathrm{P} \neq \mathrm{NP}$, $\forall L \in \mathrm{NP} \setminus \mathrm{P}$, *there is some* L' *reducible to* L *such that* $L' \notin \mathrm{P}$ *and* L *is not reducible to* L'.

Side note: If $\mathrm{P} \neq \mathrm{NP}$ then it may be undecidable whether a language $L(M)$ is decidable in P, for $L(M) \in \mathrm{NP}$. This follows from Rice's Theorem.

2 Relativization

We explore the question of whether diagonalization can help us with P versus NP. Take satisfiability as our candidate language. SAT then needs the power to simulate any poly-time language, and negate its answer. For an interesting result, we would need SAT to speed up things by some polynomial factor. In exploring this question, the logicians gave us relativization, which implies that we can't use diagonalization as the essential part of a proof that $\mathrm{P} \neq \mathrm{NP}$.

Diagonalization proofs "relativize" so that for $L \neq L_i$, $L^f \neq L_i^f$ for all f. They found f_i such that $\mathrm{P} \neq \mathrm{NP}$ relative to the oracle, and also f_2 such that they were equal.

Theorem 3 (Baker, Gill, Solovay)

$$\exists f_1 : \mathrm{P}^{f_1} \neq \mathrm{NP}^{f_1}$$

$$\exists f_2 : \mathrm{P}^{f_2} = \mathrm{NP}^{f_2}$$

Sketch of Proof We could try SAT for f_2, since this way we would have $\mathrm{NP} \subseteq \mathrm{P}^{f_2}$. But then we still have to show $\mathrm{NP}^{\mathrm{SAT}} \in \mathrm{P}^{f_2}$. In particular this would need us to show Co-Min-CNF is in $\mathrm{P}^{\mathrm{SAT}}$, which we don't know to be true.[1] Fortunately, picking f_2 to be some PSPACE-complete language (i.e. TQBF), works, because now we have $\mathrm{PSPACE} \subseteq \mathrm{P}^{f_2}$, and $\mathrm{NP}^{f_2} \subseteq \mathrm{NPSPACE} \subseteq \mathrm{PSPACE}$ and so we get $\mathrm{P}^{f_2} = \mathrm{NP}^{f_2} = \mathrm{PSPACE}$.

Finding f_1 is harder, and uses a language that is more specifically crafted to this purpose. Essentially, the authors found a problem that requires brute force search in just the right way. More details in the lecture notes. ■

Definition 4 (Oracle Turing Machine) *This type of Turing Machine has a special "Oracle" state, and an "Oracle" tape, which is write-only. When the machine goes into an Oracle state, an external oracle erases the Oracle tape and transitions into a state according to whether it accepted or rejected what was on the tape. In other words, the computational complexity of the Oracle language is treated as constant, and we don't take its execution into account.*

The Oracle tape is write-only, so that we can't "cheat" and use it as a work tape for a space-limited machine use it as a work tape.

M^f *is the computation of an Oracle machine* M *with oracle for boolean function* f.

3 Proof of Ladner's Theorem

Theorem 5 (Ladner's Theorem) *If* $\mathrm{P} \neq \mathrm{NP}$, $\forall L \in \mathrm{NP} \setminus \mathrm{P}$, *there is some* L' *reducible to* L *such that* $L' \notin \mathrm{P}$ *and* L *is not reducible to* L'.

(The proof below is significantly expanded from the original version.[2])

Proof

[1] Thanks to Michael Forbes for pointing out an error in a previous version of these notes.

[2] Thanks to Asilata Bapat for raising concerns about the previous proof.

Notational abuse: For string x and language L, we let $L(x) = 1$ if $x \in L$ and 0 otherwise. We will suppress the difference between machines M and the language $L(M)$ decided by them, and use $M(x)$ to denote $L(M)(x)$. Reductions will be denoted M^O, where O denotes a generic oracle. Thus the machine obtained by applying reduction M^O with an oracle for language L will be denoted M^L.

Overview: We will define the language L' by giving a polynomial time (many-one) reduction M^O such that $L' = L(M^L)$. This will immediately imply $L' \leq L$. We will work with the reduction to ensure that $L' \notin \mathrm{P}$ and $L \not\leq L'$.

To this end let $M_1^O, M_2^O, \ldots,$ be an enumeration of all polynomial time reductions. Let $N_1, N_2, \ldots$ be an enumeration of all polynomial time algorithms. We will try to make sure that $L \neq M_j^{L'}$ for every j; and also that $L' \neq N_j$ for every j.

Our rough intent is to define L' in stages j for $j = 1, 2, 3, \ldots$. Within each stage there will be two substages $j.1$ and $j.2$. In stage $j.1$, L' will look like a P language (say the empty language), and in stage $j.2$, L' will look like L. If we could do this reliably we would ensure that in stage $j.1$ we rule out the possibility that $L = M_j^{L'}$ and in stage $j.2$ we rule out the possibility that $L' = N_j$. However it is tricky to verify whether this possibility is ruled out yet or not, and so we adopt a "lazy" strategy which will compare L vs. $M_j^{L'}$ (and L' vs. N_j) on very small instances to see if they are equal, and stop when time runs out. When the time runs out, if we are in some stage of the form $j.1$, we let $L'(x) = 0$, and otherwise we let $L'(x) = L(x)$. Our analysis will show that this will lead to a language L' of the desired form.

Construction of L': On input x, do the following:

1. In the first phase, run the following procedure for $|x|/2$ units of time:
   - For $j = 1$ to ∞ do
     - In Stage 1, Enumerate all strings $y = \lambda$, 0, 1, 00, ... till we find a y such that $L(y) \neq M_j^{L'}(y)$. When you find such a y move to Stage 2.
     - In Stage 2, Enumerate all strings $z = \lambda$, 0, 1, 00, ... till we find a y such that $L'(z) \neq N_j(z)$. When you find such a y, finish this stage, increment j and go back up to Stage 1.

   Let procedure be in Stage i (for $i \in \{1, 2\}$) when $|x|/2$ units of time conclude.
2. In the second phase, if $i = 1$, let $L'(x) = 0$, else let $L'(x) = L(x)$.

Analysis of L' We start by noticing (despite the recursive definition of L') that $L'(x)$ can be computed in time $|x|/2 + O(1)$ (for the first phase) with at most one oracle call to L (for the second phase). Thus $L' \leq L$ as desired. Note further that this also places L' in NP and hence in exponential time.

Now we claim that for every j, there is a string y such that $L(y) \neq L(M_j^{L'})(y)$, and a string z such that $L'(z) \neq L(N_j)(z)$. Assume otherwise and let j be the smallest index for which this assertion is not true.

First consider the case that $L(y) = L(M_j^{L'})(y)$ for every y. Since j is the smallest index for which the assertion is not true, we have that there exist strings $y_1, \ldots, y_{j-1}$ and $z_1, \ldots, z_{j-1}$ such that for every $k < j$, $L(y_k) \neq L(M_k^{L'})(y_k)$ and $L'(z_k) \neq L(N_k)(z_k)$. Let m be the length of the longest string among $y_1, \ldots, y_{j-1}$ and $z_1, \ldots, z_{j-1}$. We note that for every string x of sufficiently long length (think $|x| > 2^{2^m}$), $|x|/2$ units of time is sufficient to compute $L(y)$ and $M_k^{L'}(y)$, $L'(z)$ and $N_k(z)$ for every $k < j$ and every y, z of length at most m. Thus, on input x, the algorithm for deciding L' above will find the strings $y_1, \ldots, y_{j-1}$ and $z_1, \ldots, z_{j-1}$ that allow it to finish previous stages and take it to Stage $j.1$. At this stage, since $L'(y) = L(M_j^L)(y)$ for every y it will get stuck and so will output $L'(x) = 0$. Thus we get that L' only contains strings of finite length (less than "2^{2^m}"), and hence is in P; but on the other hand $L \leq L'$, since $L = M_j^{L'}$. But this implies L is also in P which contradicts the hypothesis that $L \in \mathrm{NP} - \mathrm{P}$.

The case where there exists y_j such that $L(y_j) \neq M_j^{L'}(y_j)$ but $L'(y) = N_j(y)$ for all y is similar. In this case $L'(x) = L(x)$ for all but finitely many x and $L' \in \mathrm{P}$ (since N_j decides it), again leading to a contradiction. ■

4 Hierarchy Theorems

We return to the following result, and attempt to improve on its granularity.

$$\text{SPACE}(f(n)) \subsetneq \text{SPACE}(\omega(f(n)))$$

It turns out that it is hard to keep track of space at a small level in a meaningful way.

Theorem 6 (Speedup Theorem) [3]

$$\forall c \ \text{SPACE}(f(n)) \subseteq \text{SPACE}(f(n)/c)$$

Anything that you can do in space $f(n)$ can also be done in space $f(n)/c$, for any constant c. This measure of space corresponds to cells on the work tape. Similarly, for time:

$$\text{TIME}(t(n)) \subseteq \text{TIME}(t(n)/100 + n)$$

Essentially, we achieve these results by increasing the alphabet and control size ("increase CPU size"). In summary, it is hard to talk meaningfully about constant factors for complexity classes

4.1 Next Time

Circuits as a model of computing. Given a gate set, we look at what we can build.

[3] Thanks to Brendan Juba for pointing out that this theorem is not due to Blum (as claimed in the lecture), who gave a different speedup theorem.