18.405J/6.841J Advanced Complexity Theory	February 25, 1997
Lecture 6	
Lecturer: Dan Spielman	*Scribe: Yevgeniy Dodis*

1 Diagonalization and NP

The main result of this lecture is the following somewhat surprising theorem:

Theorem 1 *There are (computable) oracles A and B such that $P^A = NP^A$, but $P^B \neq NP^B$.*

Before proving the theorem, let's make a remark of its importance. A lot (most) of "inequality results" (i.e. the results stating that 2 classes are different) in complexity are proven using the *diagonalization method.* We somehow construct an object of one class that is guaranteed to be different from all objects of the second class, thus proving that the classes are different. A typical example of this approach is unsolvability of the Halting problem. Another one is time/space hierarchy theorem, where a "stronger" machine simulates any weaker one for a particular input (number of that machine) and reverses the output of the "weaker" machine for that particular input. It was believed initially that this powerful and common diagonalization idea could be used in proving that $P \neq NP$. The above result shows that diagonalization is very unlikely to help in this problem. The intuitive reason for that is that if diagonalization works to prove inequality of non-relativized classes P and NP, we expect it to prove inequality of P^A and NP^A, which are equal. On the other hand, even in a very unlikely case that $P = NP$, we don't expect the proof of this fact to be a direct simulation of any non-deterministic machine by a deterministic one, since that simulation is likely to work to prove equality of P^B and NP^B, which are not equal. In essence, it the theorem is a further evidence that $P = NP$ question is hard to prove.

Now, let's go to the proof of the theorem.

a) Let's construct required language A, s.t. $P^A = NP^A$.

IDEA: we need an oracle, for which non-determinism "does not matter". We know that $PSPACE = NPSPACE$, suggesting that we can try any $PSPACE$-complete language as a good candidate for A.

PROOF: Let $A = QBF$ (quantified boolean formulas language). We claim that $PSPACE \subseteq P^A$. Indeed, if $L \in PSPACE$, we construct M^A as follows: since QBF is $PSPACE$-complete, map instance of problem in L to equivalent instance of QBF using the polynomial time reduction, and then simply ask the oracle QBF for an answer. Next, we clearly have that $P^A \subseteq NP^A$. Finally, we claim that $NP^A \subseteq NPSPACE$. Our $NPSPACE$ machine simply simulates the NP^A machine, and whenever our NP^A-machine needs to ask the QBF-oracle, the $NPSPACE$ machine simply computes the answer by brute force in $PSPACE$, which it can do since $QBF \in PSPACE$. And recalling that $PSPACE = NPSPACE$, we get a chain of inclusions: $PSPACE \subseteq P^A \subseteq NP^A \subseteq NPSPACE = PSPACE$, making $P^A = NP^A$.

b) Now, let's construct oracle B, such that $P^B \neq NP^B$. We want to construct a language $L \in NP^B - P^B$. Clearly, L should crucially depend on B. For any oracle B, define

$$L(B) = \{x | \exists y \in B s.t. |y| = |x|\},$$

so $L(B)$ contains all strings of the same length as some element of B. First, it's easy to see that for any B we have $L(B) \in NP^B$. Indeed, we just guess an appropriate string $y \in B$ of same length as input x and ask our oracle B whether $y \in B$.

We will construct B in such a way that $L(B) \in NP^B - P^B$, which by above remark is equivalent to saying that $L(B) \notin P^B$. To decide $L(B)$ deterministically in polynomial time, some polynomial oracle TM M^B, given an input x of size n, should be able to "go through" 2^n possible choices for y in polynomial time, and so, it has to do it without going through all the choices for big enough n. But this suggests a way to construct B. B will "deceive" any polynomial time oracle TM $M^?$ (? stands for the fact that M uses some, yet unspecified, oracle) in the following way: we choose large enough n, so that $M^?$ runs in less that 2^n time, and so asks less than 2^n questions to the oracle. We feed $M^?$ input 0^n, and answer "no" to all queries that $M^?$ makes. If at the end $M^?$

rejects (which any reasonable machine should do when getting constant "no"s, but $M^?$ might be "stupid"), we find a string y of length n that was not queried by $M^?$ (it exists since $M^?$ asked less than 2^n queries), and declare $y \in B$, thus making $0^n \in L(B)$ and making $M^?$ be wrong on input 0^n in trying to decide $L(B)$. If, for a strange reason, $M^?$ accepts, we say that, indeed, no string of length n is in B, thus making $0^n \notin L(B)$ and making $M^?$ wrong again. This way we guarantee that $M^?$ does not decide $L(B)$. Any particular polynomial-time machine $M^?$ can be "deceived" for ANY large enough n. Now, we only need to make B deceive ALL polynomial time oracle machines. Here we use brute force, by enumerating all such machines and picking larger and larger n for successive machines to avoid overlapping (so here we are using the diagonalization method despite the fact that we want to show that it's useless in proving $P \neq NP$). Here is precise (algorithmic) description of B:

Let $M_1^?$, $M_2^?$, $M_3^?$, ... be an enumeration of all polynomial-time oracle TM's (for now, we don't care how to compute it, we are *defining* B, so we just need the existence of enumeration).

1. Let $B_0 = \emptyset$, $n = 0$

2. for $i = 1, 2, 3, \dots$ do

   (a) choose $n_i > n$, such that $M_i^{B_{i-1}}$ on input 0^{n_i} asks fewer than 2^{n_i} questions (such n_i exists).

   (b) If $M_i^{B_{i-1}}$ accepts 0^{n_i} then let $B_i = B_{i-1}$ (so we don't include any new strings to B). Else if it rejects Let $B_i = B_{i-1} \cup \{$ some word x s.t. $|x| = n_i$ and x was not asked by $M_i^{B_{i-1}}\}$.

   (c) set $n = max$(length of longest quiery asked by $M_i^{B_{i-1}}$, on input 0^{n_i}.

3. set $B = \bigcup_{i=0}^{\infty} B_i$

From above remarks we see that $L(B) \in NP^B - P^B$, as M_i^B gives incorrect answer for input 0^{n_i}.

Remark: Above description of B was mathematical, but we can make B computable if we modify a bit the above algorithm. Instead of enumerating all polynomial oracle TM's, we can enumerate only the oracle TM's with explicit polynomial timer that will stop the machine if it runs for too long. This will include eventually all different (i.e. not giving same answer on all inputs) polytime oracle TM's. Then we can pick the smallest n_i satisfying the criteria we want. Same for picking a string x to be included in B when we extend B with a new string of size n_i. Then to decide whether $x \in B$ we run the above algorithm until $n_i \geq |x|$ and see whether x was included into B. This makes B computable.

2 Randomized Complexity Classes

Very often a brute force deterministic approach to a problem requires examining a lot (or all) of the possibilities, which is very costly to do. On the other hand, we might be sure that a good fraction of the possibilities (say a half) will be "good", i.e. they will provide the answer we want, we just don't know how to organize the search in the deterministic manner to be sure that we soon encounter such a good choice. Of course, in theory we can use a non-deterministic guess, but that's not realistic. On the other hand, if we have a good source of random numbers, we can choose a possibility at random and have a good chance to succeed. Contrary to non-deterministic approach, this one is very realistic and quite powerful. A motivating example to show the power of randomness (that will lead directly to the definition of randomized complexity classes) is the problem of polynomial identities. We are given 2 multivariable polynomials in some (maybe different for the two) implicit form (say, one is a determinant, the other is a product of multinomials). Basically, we assume that the only thing we can do efficiently with either polynomial is to evaluate it at an arbitrary point (the details of other things we can do depends on what implicit form is used). We need to check whether the given multivariable polynomials p and q are identical. Let $r = p - q$.

The simplest idea that comes in mind is to evaluate p and q over some appropriately chosen set of points, so that $p \equiv q$ iff for all above mentioned points x, $p(x) = q(x)$, i.e. we want a set of points containing at least one non-root of r, if there is any. Unfortunately, r might have a lot (even infinite number) of zeros, while being not identically zero, and since we don't know much about r, we don't know how to determine whether $r \not\equiv 0$ in a systematic fashion. Here comes randomization. Choose a point at random form an appropriate domain (see below). If $r \not\equiv 0$, we should have a decent chance to catch it very-very quickly. That's the correct intuition. The details follow.

Schwartz lemma: Assume $p(x_1, \ldots, x_n)$ is a non-zero polynomial, where each variable x_i has degree at most $d > 0$. Let I be any domain of size $\geq nd$. Then

$$\Pr_{\{x_1,\ldots,x_n\}\in I^n}(p(x_1,\ldots,x_n)=0) \leq \frac{dn}{|I|}$$

Proof: We use induction on n. Let $n = 1$. A non-zero polynomial of degree $\leq d$ can have at most d roots, so when we choose x_1 from a domain of size $|I|$, we have at least $\frac{d}{|I|}$ chance to pick a non-root, exactly as inequality says.

Now, the inductive step. Let $d \geq d_1$ = highest degree of x_1 in p. And let $p_2(x_2, \ldots, x_n)$ be the coefficient in front of $x_1^{d_1}$. We have by inductive hypothesis that

$$\Pr_{\{x_2,\ldots,x_n\}\in I^{n-1}}(p(x_2,\ldots,x_n)=0) \leq \frac{d(n-1)}{|I|}$$

We have:

$$\begin{array}{rl}
\Pr_{\{x_1,\ldots,x_n\}\in I^n} & (p(x_1,\ldots,x_n)=0) \\
= & \displaystyle\Pr_{\{x_1,\ldots,x_n\}\in I^n}(p(x_1,\ldots,x_n)=0|p_2(x_2,\ldots,x_n)=0) \times \Pr_{\{x_2,\ldots,x_n\}\in I^{n-1}}(p(x_2,\ldots,x_n)=0) \\
+ & \displaystyle\Pr_{\{x_1,\ldots,x_n\}\in I^n}(p(x_1,\ldots,x_n)=0|p_2(x_2,\ldots,x_n)\neq 0) \times \Pr_{\{x_2,\ldots,x_n\}\in I^{n-1}}(p(x_2,\ldots,x_n)\neq 0) \\
\leq & \displaystyle 1 \times \frac{d(n-1)}{|I|} + \frac{d}{|I|} \times 1 = \frac{dn}{|I|}.
\end{array}$$

The first bound $\frac{d(n-1)}{|I|}$ was made by the inductive assumption for $n-1$, the second bound $\frac{d}{|I|}$ - by observing that as long as we know that $p_2(x_2, \ldots, x_n) \neq 0$, no matter what $x_2, \ldots, x_n$ are chosen, there can be at most $d_1 \leq d$ out of $|I|$ possible values of x_1 making $p(x_1, \ldots, x_n) = 0$, and so the above bound follows, and thus is the lemma.

Corollary: Let $|I| \geq 2nd$, $p(x_1, \ldots, x_n)$, $q(x_1, \ldots, x_n)$ be 2 polynomials, and each variable in either of them has maximum degree $\leq d$. If we set $r = p - q$ then Schwartz lemma gives us the following:

1. If $r \equiv 0$ (i.e. $p \equiv q$), then
$$\Pr_{\{x_1,\ldots,x_n\}\in I^n}(r(x_1,\ldots,x_n)\neq 0) = 0$$
2. If $r \not\equiv 0$ (i.e. $p \not\equiv q$), then
$$\Pr_{\{x_1,\ldots,x_n\}\in I^n}(r(x_1,\ldots,x_n)\neq 0) \geq \frac{1}{2}$$

As we'll define next lecture, above corollary means that the language of polynomial identities is in the randomized complexity class RP. Let's try to answer a big question though: can we derandomize (i.e. make deterministic) the above algorithm, and, in general, how essential is randomization? First, observe that if we make $I \geq knd$ for some k, then probability of error (i.e. $r \not\equiv 0$, but we chose a root of r as a sample point; call such misleading points "bad" for a particular identity) is $\leq \frac{1}{k}$. Now, assume we look at all possible polynomial identities that can be written using m bits. There are no more than 2^m such identities. As we observed, at most $\frac{1}{k}$ of possible sample points is bad for a particular identity, so, by union bound, at most $\frac{2^m}{k}$ fraction of sample points can be bad for *at least one* of 2^m identities. But now, if we let $k > 2^m$, say $k = 2^{m+1}$, we know that there must exist a sample point $y_1, \ldots y_n$ good for *all* identities. The magnitude of k is exponential, but it has has polynomial length, so we are OK. What it means is that we can feed above point as an advice for all identities of size m, and those identities can be trivially verified in polynomial time, so it looks like we don't need randomization. But the problem, of course, is that we'll have different sample points for different input lengths m, and we don't know how to find such points efficiently (that's why we used randomization in the first place). What we showed, however, is that the problem of polynomial identities is in $P/poly$, i.e. we have a non-uniform polynomial algorithm for deciding it. As we'll see, it's a general pattern: any problem that can be solved in polynomial time using randomization, can be solved in non-uniform polynomial time (the general proof is almost identical copy of the one we have for polynomial identities), but it remains one of the biggest questions whether we can avoid randomization altogether, i.e. whether $P = RP$.