Instructor: Andrej Bogdanov **Notes by:** Yingchao Zhao

So far in the class we encounter two notions of "hardness." When we define NP-hardness, for instance, we want to rule out algorithms that solve problems on *all* inputs. In our analysis of the Nisan-Wigderson generator, we looked into a different notion of hardness. The assumption we used there is that there are problems in EXP for which no algorithm can do essentially better than guess the answer at random.

Generally, we might expect that designing an algorithm that does slightly better than random guessing should be easier than designing one that solves the problem in question on all instances. In certain specific settings, this is indeed the case. For example, there are natural distributions on random 3SAT formulas such that for a random formula chosen from this distribution, most of the time it is possible to tell efficiently if the formula is satisfiable or not. In contrast, we do not expect this to be true for *all* formulas since 3SAT is NP-complete.

However, in other settings the opposite is true: Designing an algorithm that does slightly better than guessing the solution at random is as hard as designing an algorithm that correctly solves all instances of the problem.

Today we will begin to look at some examples of such problems. To simplify the exposition, it helps to think of these two notions of hardness – getting everything correct and doing slightly better than random guessing – as two extremes of *average-case hardness*. We can also look at other degrees of average-case hardness: Can we solve a given problem on 90%, 99%, or even $1 - 1/n$ fraction of inputs of length n?

1 The permanent

Our first example of a problem which is as hard to solve in the worst case as it is in the average case is the permanent. To define the permanent, we go back to the #P-complete problem of counting perfect matchings in a bipartite graph. Given a bipartite graph $G = (U, V, E)$ with $|U| = |V| = n$, the adjacency matrix A of G is the $n \times n$ matrix where $A_{i,j} = 1$ if (i, j) is an edge in G (where $i \in U, j \in V$) and $A_{i,j} = 0$ otherwise. With A, we can rewrite the number of perfect matchings in G as

$$\text{number of perfect matchings} = \sum_{\pi} \prod_{i=1}^{n} A_{i,\pi(i)}$$

where π ranges over all permutations of the set $\{1, \ldots, n\}$.

We can now think of this quantity as a function of the matrix A rather than a function of the graph G. Here, the matrix A takes $0 - 1$ values, but generally A can take arbitrary integer values, or values in any finite field $\mathbb{F}$. Given an $n \times n$ matrix A with entries in a finite field $\mathbb{F}$, we define the

permanent of A over $\mathbb{F}$ as

$$\mathrm{per}(A) = \sum_{\pi} \prod_{i=1}^{n} A_{i,\pi(i)}$$

where sums and products are over the finite field $\mathbb{F}$. Since counting perfect matchings is #P complete, there is no polynomial-time algorithm for computing the permanent over the integers or fields of characteristic $> n!$ unless $\mathrm{P}^{\#\mathrm{SAT}} = \mathrm{P}$. In fact, even if we could compute the permanent efficiently over fields of size $\mathrm{poly}(n)$, then $\mathrm{P}^{\#\mathrm{SAT}} = \mathrm{P}$.

Now let us look at the average-case version of the permanent problem: For what fraction of $n \times n$ matrices A can we compute $\mathrm{per}(A)$? It turns out that computing $\mathrm{per}(A)$ for "most" matrices A is no easier than doing so for *every* matrix A:

Theorem 1. *If there is a randomized polynomial-time algorithm A such that $\Pr_X[A(X) = \mathrm{per}(X)] > 1 - 1/(3n+3)$, where X is chosen at random from the set of $n \times n$ matrices over $\mathbb{F}$, where $\mathbb{F}$ is a field of size at least $n+2$, then permanent over $\mathbb{F}$ can be computed on* all *inputs in randomized polynomial time.*

By the assumption of the theorem, we are given access to algorithm A that computes per correctly on a $1 - \frac{1}{3n+3}$ fraction of inputs. Now we want to compute per on an arbitrary input X. The output $A(X)$ might be correct or incorrect, and we don't even know which is the case. The idea is to reduce the computation of $\mathrm{per}(X)$, for an arbitrary matrix X, to running the algorithm A on a few random inputs. Since A behaves well on random inputs, this algorithm should compute $\mathrm{per}(X)$ for arbitrary X with high probability.

How do we reduce the problem of computing permanents of arbitrary matrices to computing permanents of random matrices? We look at per as a function in the n^2 variables A_{ij}. The main property we will use is that per is a polynomial of degree n in these variables. In fact, the argument we show will apply to any polynomial of sufficiently small degree, so we will switch to the following more abstract setting:

We have a polynomial $p : \mathbb{F}^n \to \mathbb{F}$ of degree d, and we are given access to an "oracle" function $A : \mathbb{F}^n \to \mathbb{F}$ that computes p on a $1 - \frac{1}{3d+3}$ fraction of inputs. Now we want to compute p on arbitrary input x using A. Consider the following algorithm B:

B: On input x,
Choose random $y \in \mathbb{F}^n$.
Let T be an arbitrary subset of $\mathbb{F}$ of size $d+1$ not containing 0.
For each $t \in T$, compute the value $a_t = A(x + ty)$.
Compute the unique polynomial $q(t)$ of degree d such that $q(t) = a_t$ for $t \in T$:
$q(t) = \sum_{i \in T} a_i \cdot \big(\prod_{j \in T - \{i\}} (t-j)\big) / \big(\prod_{j \in T - \{i\}} (i-j)\big)$.
Output $q(0)$.

To see what is going on, let us first imagine that $A(x) = p(x)$ for all inputs $x \in \mathbb{F}^n$. For fixed x and y, consider the univariate polynomial $r(t) = p(x + ty)$: Since p is a polynomial of degree d, this is a polynomial of degree (at most) d as well. One way to specify $r(t)$ is to give the values of r

on any collection T of $d+1$ distinct points. Then we can recover the value of r at any other point by interpolation: First, compute the unique polynomial q of degree d which is consistent with the values of r on T and then output the value of this polynomial at the desired point. In particular, by doing so we can recover $q(0) = r(0) = p(x)$, which is what we were supposed to compute.

In our setting, $A(x) = p(x)$ not for all but only for "most" of the inputs x. The reason the above argument can still be carried out is the following: No matter what x is, for a random y, the inputs $x + ty$ are uniformly random in $\mathbb{F}^n$, so at the points that are queried by B with high probability A will look exactly like p. More precisely:

Claim 2. *For every x, $\Pr_{y \sim \mathbb{F}^n}[\forall t \in T : q(t) = p(x + ty)] > \frac{2}{3}$.*

Proof. We have

$$\Pr[\exists t \in T : p(x + ty) \neq q(t)] \leq \sum_{t \in T} \Pr[p(x + ty) \neq q(t)] \leq (d+1) \cdot \frac{1}{3(d+1)} = 1/3$$

because for every x and every $t \in T$, $x + ty$ is a uniformly random point in $\mathbb{F}^n$. □

Now conditioned on $q(t) = p(x + ty)$ for all $t \in T$, we argue as before that $q(0) = p(x)$. This follows from the fact that for any fixed choice of x and y, the function $r(t) = p(x + ty)$ is a univariate polynomial of degree d. Since $q(t) = a_t = r(t)$ for all $t \in T$ and both q and r are degree d polynomials, it follows that $q(t) - r(t)$ is a degree d polynomial that evaluates to zero at $d+1$ distict points, so it must be that $q(t) - r(t)$ is the zero polynomial. Then $B(x) = q(0) = r(0) = p(x)$.

To summarize, we showed an algorithm B such that for any input x, $B(x) = p(x)$ with probability at least $2/3$ over the internal randomness of B. Applying this algorithm to the function $p(X) = \mathrm{per}(X)$ proves Theorem 1.

2 Worst-case to average-case reductions for E

Recall that in our proof of the Nisan-Wigderson Theorem we assumed the existence of decision problems in E (deterministic time $2^{O(n)}$) which are hard to compute by circuits of size $2^{\delta n}$ on $1/2 + 1/2^{\delta n}$ fraction of inputs of length n. We also mentioned that the same consequence can be obtained under the weaker assumption that there are problems in E which are hard to compute in the worst case by circuits of the same size. One way to obtain this stronger form of the theorem is to start from the weaker one and apply the following general result of Impagliazzo and Wigderson:

Theorem 3. *Suppose there is a function in* E *such that for some $\delta > 0$ and every circuit family C of size $2^{\delta n}$, $C(x) \neq f(x)$ for some input x. Then there is a function $f \in$ E such that for some $\delta > 0$ and every circuit family C of size $2^{\delta n}$, $\Pr_{x \sim \{0,1\}^n}[C(x) = f(x)] < 1/2 + 2^{-\delta n}$ for sufficiently large n.*

This is an example of the same phenomenon of worst-case and average-case hardness equivalence as what we saw for the permanent. We won't prove this theorem, but for now we will show the following:

Theorem 4. *Suppose that* $\mathrm{E} \not\subseteq \mathrm{BPP}$. *Then there is a function* $f \in \mathrm{E}$ *such that for every polynomial-size circuit family* C, $\Pr_{x \sim \{0,1\}^n}[C(x) = f(x)] = 1 - 1/(3n+3)$ *for sufficiently large* n.

In the next lecture we will discuss one way to improve the quantity $1 - 1/(3n+3)$ above to $1/2 + \epsilon$ for any constant ϵ.

To prove Theorem 4, we try to imitate the proof of Theorem 1. The crucial property we used there is that permanent is a polynomial of low degree in its inputs. Do problems in E also admit a representation as low-degree polynomials? Indeed this is so: Given any decision problem $L \in \mathrm{E}$, let $g : \mathbb{F}^n \to \mathbb{F}$ be the multilinear extension of $L(x)$, where $x \in \{0,1\}^n$ (let us assume for convenience that $\mathbb{F}$ has characteristic 2 and size $2^{\log n + 1}$). This f is a polynomial of degree at most n. Then for any input $x \in \mathbb{F}^n$, $g(x)$ can be computed in time $2^{O(n)}$ via the interpolation formula which queries L at all points in $\{0,1\}^n$.

To obtain a decision problem in E, we need to turn this polynomial $g : \mathbb{F}^n \to \mathbb{F}$ into a decision problem $f : \{0,1\}^{m(n)} \to \{0,1\}$. To do so, we represent each element in $\mathbb{F}$ by some binary string of length $\log n + 1$ in a canonical way, and define

$$f(x,i) = g(x)_i = \text{the } i\text{th bit of } g(x).$$

Here, $x \in \mathbb{F}^n \cong \{0,1\}^{n(\log n+1)}$ represents the input to g, and $i \in \{1, \ldots, \log n + 1\} \cong \{0,1\}^{\log(\log n+1)}$ indexes one of the bits in $g(x)$. It is not hard to check that if $L \in \mathrm{E}$, then $f \in \mathrm{E}$ as well.

We now show that if f can be computed on a $1 - 1/(3m+3)$ fraction of inputs of length m by some polynomial-tome randomized algorithm A, then $L \in \mathrm{BPP}$. Like the construction, the reduction has two stages: First, we give an algorithm B that computes g correctly on $1 - 1/(3n+3)$ fraction of its inputs, and then using B we construct a randomized algorithm C for L.

To obtain C from B, we proceed in exactly the same way as in the proof of Theorem 1: Since g is a polynomial of degree n over a field of size $> n + 2$, we can transform B into a randomized algorithm C that computes $g : \mathbb{F}^n \to \mathbb{F}$ correctly on all inputs. In particular, C computes L when the input is in $\{0,1\}^n$ because g is an extension of L.

It remains to show how to obtain B from A. There is only one natural way of doing so: On input $x \in \mathbb{F}^n \cong \{0,1\}^{n(\log n+1)}$, B outputs the field element whose ith bit is $A(x,i)$. Since $\Pr_{x,i}[A(x,i) = f(x,i)] > 1 - 1/(3m(n)+3)$, we have that

$$\begin{aligned}
\Pr_{x \sim \mathbb{F}^n}\left[A(x,i) = f(x,i) \text{ for all } i\right] &= 1 - \Pr_{x \sim \mathbb{F}^n}\left[A(x,i) \neq f(x,i) \text{ for some } i\right] \\
&\geq 1 - \Pr_{x \sim \mathbb{F}^n}\left[(\log n + 1) \cdot \Pr_i[A(x,i) \neq f(x,i)]\right] \\
&= 1 - (\log n + 1) \cdot \Pr_{x,i}[A(x,i) \neq f(x,i)] \\
&= 1 - \frac{\log n + 1}{3m(n)} \\
&> 1 - \frac{1}{3n+3}.
\end{aligned}$$

Therefore, $B(x) = g(x)$ for at least $1 - 1/(3n+3)$ fraction of inputs $x \in \mathbb{F}^n$, and so $C(x) = L(x)$ on arbitrary x with probability $2/3$.