CS221: Computational Complexity Prof. Salil Vadhan

Lecture 27: Average-Case Complexity, Low-Degree Extensions

11/27 *Scribe: Saurabh Sanghvi*

Contents

Today we will finish up our coverage of average-case complexity with a proof of the theorem stated last time, which effectively gave an example of a problem that is hard in the average-case if it is hard in the worst-case.

1 Recap

Last time, we defined the problem Modular Permanent as follows:

- Input: $(1^k, 1^p, M)$ p prime, $p > k+1$, $M \in \mathbb{Z}_p^{k\times k}$ (So $k \times k$ matrix with entries in $\mathbb{Z}_p$).
- Output: $\mathrm{Perm}(M) \bmod p$

We are in the process of relating the worst-case complexity of this problem to its average-case complexity under the distribution $\mu_{k,p}$ — the uniform distribution over $M \in \mathbb{Z}_p^{k\times k}$.

Theorem 1 *If there exists a probabilistic polynomial time (PPT) algorithm A such that* $\Pr_{M\leftarrow\mu_{k,p}}[A(1^k, 1^p, M) \neq \mathrm{Perm}(M) \bmod p] \leq \frac{1}{3(k+1)}$, *then there exists a PPT algorithm B such that for* every $M \in \mathbb{Z}_p^{k\times k}$, $\Pr_{\textit{coins of } B}[B(1^k, 1^p, M) \neq \mathrm{Perm}(M) \bmod p] \leq 1/3$

In particular, we'd have that ModPerm $\in$ **BPP**, and thus $\mathbf{P}^{\#\mathbf{P}} = \mathbf{BPP}$. (We proved ModPerm was $\#\mathbf{P}$-hard last class). Note that this theorem is usually read negatively—since we don't believe these consequences hold, it is unlikely such an A exists. The proof of this theorem follows (as shown in the last class), from this more general one, which we will now prove:

2 Main Result

Theorem 2 *For $\mathbb{F}$ finite field, $|\mathbb{F}| \geq d+2$, given oracle access to $f : \mathbb{F}^m \to \mathbb{F}$ disagreeing with some polynomial $p : \mathbb{F}^m \to \mathbb{F}$ of degree $\leq d$ in fewer than $\frac{1}{3(d+1)}$ fraction of points in $\mathbb{F}^m$, one can compute p everywhere with high probability in time* $\mathrm{poly}(d, m, \log|\mathbb{F}|)$.

Proof: The basic method we will use is called **random self-reducibility**—the process of taking an algorithm that works on random points and making it work at an arbitrary point.

More specifically, what is going to happen here is that we have a polynomial defined over an $\mathbb{F}^m$ hypercube, and we want to evaluate the polynomial at any given point, using an oracle that errs on a certain percentage of them. The strategy is to pick a random line coming out of the point. Since the points on this line are uniformly random, with high probability we'll be able to evaluate the polynomial at sufficiently many points along the line. But since they are still related in this close way to the original point, using the value of the polynomial at those points we can interpolate and get the polynomial at the desired point.

Use the following algorithm to compute $p(x)$, given oracle for f:

1. Choose $y \stackrel{\mathrm{R}}{\leftarrow} \mathbb{F}^m$.
2. Define the line $\ell : \mathbb{F} \rightarrow \mathbb{F}^m$, $\ell(t) = x + ty$.
3. Query $f(\ell(1)), \ldots f(\ell(d+1))$.
4. Interpolate to get a univariate polynomial $q(t)$ of degree $\leq d$ such that $q(i) = f(\ell(i))$ for $i = 1, \ldots d+1$.
5. Output $q(0)$.

Before we formally analyze this proof, one should note the use of two key observations early on that gave us this algorithm:

1. The first is noting that $p(\ell(t))$ is a univariate polynomial of t that has degree d. This is clear because p is a polynomial of n variables with degree at most d, and ℓ is linear—and so plugging in ℓ will not increase the degree. This is critical because basically we are trying to capture this polynomial with q to get $p(x)$, without knowing p in general, and so we need it to have a particular low degree.
2. Second, for all $i \in \mathbb{F} \setminus \{0\}$, $\ell(i)$ is uniformly random in $\mathbb{F}^m$. Granted, since each point lies on the same line they are very correlated and not independent, but still y is a completely random direction, and so each point $x + ty$ is random. This property is critical to ensuring the likelihood that we can get $p(\ell(t))$.

Another note: the choice of $1, \ldots, d+1$ as the points to be queried in step 3 was arbitrary—we just needed $d+1$ distinct points in the field.

To analyze this, we want to bound the probability this algorithm fails—i.e., $\Pr[q(0) \neq p(x)]$. The following does just that:

$$\begin{aligned}
\Pr[q(0) \neq p(x)] &\leq \Pr[\exists i \in \{1, \ldots, d+1\} \text{ such that } f(\ell(i)) \neq p(\ell(i))] && (1)\\
&\leq (d+1) \cdot \frac{1}{3(d+1)} && (2)\\
&= 1/3
\end{aligned}$$

(1) clearly holds because if in fact for all i, $f(\ell(i)) = p(\ell(i))$, then clearly we will succeed because we will be able to recover $p(\ell(t))$ (by Observation 1) and so will correctly calculate $p(x) = p(\ell(0))$. Thus, the probability of failure is at most the probability that one $f(\ell(i))$ is wrong.

What is this probability for a given i? Since each of the $\ell(i)$ are uniformly distributed (by Observation 2), the chance that a particular i fails is at most the chance f errs on a random point: $\frac{1}{3(d+1)}$. Using the union bound, we multiply this chance for the $d+1$ points that could cause the failure, and we get (2).

■

There are several ways to view this result:

1. As mentioned before, this result is a way to prove that a problem is <u>hard on average</u>—the result states that if the problem is easy on average it is easy in the worst case, so by contrapositive, if it is hard in the worst case (which we usually think to be the case in these situations) then it is hard on average.

2. More "constructively," one could see this as a <u>program corrector</u>. Suppose we write a program that is somewhat buggy—and it gets some answers wrong, but is correct on most inputs. We can use this algorithm to always extract the correct answer.

3. In a similar vein, this algorithm can be seen as the <u>decoding algorithm</u> for a certain error-correcting code (the so-called Reed-Muller code). This provides a way to encode information such that even if information is corrupted in some places one can still retrieve the information. Specifically, if you've encoded some data as a low degree polynomial, then we can apply this result to recover from corruptions. The novel feature of this decoding algorithm (as compared to ones traditionally studied in coding theory) is that any single value of the polynomial can be decoded without even reading the entire codeword (rather just probing it in a few randomly chosen places).

Note: The theorem can be strengthened to recover from error probability $1/2 - \epsilon$, using a more complicated algorithm.

3 Multilinear Extension

A natural question to ask here is how general this theorem is. We saw it work for $\#\mathbf{P}$, but how does this work for other complexity classes? As it turns out, all natural classes above $\#\mathbf{P}$ which are closed under complement have complete problems with this property.

Theorem 3 (multilinear extension) *Given some finite field $\mathbb{F}$, for any $f : \{0,1\}^n \to \{0,1\}$, there is a* ***unique multilinear*** *function $\hat{f} : \mathbb{F} \to \mathbb{F}$ such that $\hat{f}|_{\{0,1\}^n} \equiv f$ (i.e., $\forall x \in \{0,1\}^n \hat{f}(x) = f(x)$). Moreover $\hat{f} \in \mathbf{FPSPACE}^f$ (in fact, $\hat{f} \in \mathbf{P}^{\#\mathbf{P}^f}$).*

Recall that a multilinear function is one that has degree 1 in each variable (and therefore has total degree at most n).

This theorem is thus a way to convert *any* problem to a low-degree polynomial, so that the previous analysis will apply. Pictorially, the theorem takes a function defined on a small subcube (the field restricted to coordinates in $\{0,1\}$), and then extends it to a polynomial defined on all $\mathbb{F}^m$. We can then apply the theorem and it will be meaningful for the points of $\hat{f}$ that we care about.

Corollary 4 *If* **PSPACE** $\neq$ **BPP** *(or* **EXP** $\neq$ **BPP***), then* **PSPACE** *(respectively,* **EXP***) has a problem which is* ***hard on average*** *with respect to the uniform distribution. (That is, there will not be a PPT algorithm A solving the problem with error $\leq \frac{1}{p(n)}$ for some polynomial p).*

Proof:

Let $f \in \mathbf{PSPACE} \backslash \mathbf{BPP}$. Then $\hat{f} \in \mathbf{PSPACE} \backslash \mathbf{BPP}$: $\hat{f} \in \mathbf{PSPACE}$ because $\hat{f} \in \mathbf{PSPACE}^f$, and $\hat{f} \notin \mathbf{BPP}$ because f reduces to $\hat{f}$. This means that $\hat{f}$ is hard in the worst case.

But if there were to exist PPT algorithm A such that $\Pr_x[A(x) \neq \hat{f}(x)] \leq \frac{1}{3(n+1)}$, then by theorem 2 we would have that $\hat{f} \in \mathbf{BPP}$. Contradiction—and so $\hat{f}$ is hard on average.

(Note that the application of theorem 2 requires that the degree of $\hat{f}$ is at most n.)

$\blacksquare$

So with this corollary we now know how to relate average case with uniform distribution to the worst case, for $\#\mathbf{P}$ and above. (The corollary won't work with anything lower, of course, because then we won't be able to conclude that $\hat{f}$ is in that lower class, since at best we know $\hat{f} \in \mathbf{P}^{\#\mathbf{P}^f}$).

A reasonable question to ask here is: what about **NP**? As mentioned before cryptography is very interested in the average case, and so a similar result for **NP** would be most useful. Unfortunately, this is still an open problem. Ajtai in 1997 gave a worst-case/average-case equivalence for **lattice problems** (given an n-dimensional lattice L, find an approximately shortest vector in L), which are in **NP** but not known to be **NP**-complete (but the problems still appear to be hard—they are not known to be in **BPP**).

Looking back over these results, two points to take away are the important roles played by **algebra** and **randomization**:

1. Algebra was found to be quite useful as computational problems and complexity classes can be encoded using polynomials, which are themselves tools of algebra. The benefit of doing this is that we can put the problem into an area where we have much more structure and machinery to prove results. We've seen this many times before: the result that PERMANENT is $\#\mathbf{P}$-complete gives a lot of structure to $\#\mathbf{P}$, we used polynomials to prove that PARITY $\notin \mathbf{AC}_0$, and on the problem set you use polynomials to show that testing equivalence of read-once branching programs is in **BPP**.

2. Similarly, randomization was crucial above: only through randomly picking that line were we able to bootstrap an algorithm that gets the polynomial right on a few points to one that gets it right with high probability on all points. In general, randomization seems to be very powerful in conjuction with algebra — intuitively because properties of polynomials tend to be reflected almost everywhere (and hence at random points).

Needless to say, these two tools show up together in many more parts of complexity theory, as we will see.