6.841 Advanced Complexity Theory April 7, 2009

Lecture 16

Lecturer: Madhu Sudan *Scribe: Paul Christiano*

1 Worst-Case vs. Average-Case Complexity

So far we have dealt mostly with the worst-case complexity of problems. We might also wonder aobut the average-case complexity of a problem, namely, how difficult the problem is to solve for an instance chosen randomly from some distribution. It may be the case for a certain problem that hard instances exist but are extremely rare. Many reductions use very specifically constructed problems, so their conclusions may not apply to "average" problems from a class.

For example, consider the problem of sorting a list of integers. We can define the worst-case complexity:

$$T(n) = \min_{A} \max_{|x|=n} \{\text{Running time of } A(x)\}$$

where the minimum is taken over all algorithms A such that $A(x)$ always outputs the list x sorted in increasing order. If we have a distribution D over the inputs we can define the average case complexity the same way, except that the minimum is taken over all algorithms A such that $A(x)$ outputs a sorted list except on a set of small probability. We can define the worst case and average case complexity of any other language or function in the same way.

We could also define several slightly different notions of average case complexity. For example, we could only consider algorithms A which are correct on all inputs but we could measure the expected running time instead of the running time on the worst instance. Notice that this is a strictly weaker notion: if we have an algorithm running in expected polynomial time, you can convert it to one which always runs in polynomial time but which is occassionally wrong by timing your algorithm and terminating it after it has run for a polynomial number of steps (for an appropriately chosen polynomial). We will always deal with our stronger definition.

Worst case complexity is a very useful measure for constructing reductions: we are allowed to reduce to instances of an arbitrarily rare special form in order to make a statement about worst case complexity. If we are dealing with average case complexity, reductions become much more difficult to describe. Another disadvantage of average case complexity is that it is only defined when we have a probability distribution over the inputs, while there isn't always a natural way to define a distribution. For example, if we want to consider the average case complexity of 3SAT we could use the distribution which chooses one clause

per variable uniformly at random. But if we did this then simply outputting "unsatisfiable" would almost always be correct, and the average case complexity of 3SAT would be constant.

Nevertheless, for some problems, such as the permanent, we are able to obtain clean results for natural distributions.

2 The Complexity of the Permanent

Preliminaries

Recall

$$\mathrm{Perm}(M) = \mathrm{Perm}\left(\{m_{ij}\}\right) = \sum_{\sigma} \prod_{i \in [n]} m_{i\sigma(i)}$$

where the sum is taken over permutations σ of $[n]$. If the entries of M are in $\{0, 1\}$ then we can form the bipartite graph of which M is the adjacency matrix: one partition of this graph contains the rows of M and the other contains the columns. There is an edge between row i and a column j iff $m_{ij} = 1$. $\mathrm{Perm}(M)$ counts the number of perfect matchings in the graph associated to M: this is clearly in #P.

If we think of the matrix entries M_{ij} as indeterminates in some field $\mathbb{F}_p$, then $\mathrm{Perm}(M)$ is a polynomial of degree n in n^2 variables. This gives us a lot of nice algebraic properties of the permanent. It also gives us intuition about how to find difficult instances of the permanent: while combinatorial problems often have a very small set of very difficult instances, for algebraic problems typically almost all of the instances are equally difficult (except for some low-dimensional space of degenerate cases). Thus we expect a random instance of permanent to be about as hard as an adversarially chosen instance.

Theorem 1 (Lipton 89; Beaver,Feigenbaum 88) *Because permanent is a low degree polynomial, it is as hard on a random instance as it is in the worst case.*

More precisely, consider the following two problems.

Worst case: given an adversarially chosen n-bit prime p and a matrix M in $\mathbb{Z}_p^{n \times n}$, compute $\mathrm{Perm}(M) \in \mathbb{Z}_p$.

Average case: given an n-bit prime p chosen uniformly at random and a matrix M chosen uniformly at random from $\mathbb{Z}_p^{n \times n}$, compute $\mathrm{Perm}(M) \in \mathbb{Z}_p$.

We make the following claim: you can turn an algorithm A which solves the average case problem with probability $1 - \delta \geq 1 - \frac{1}{3n}$. (over the choice of problem instance) into a BPP algorithm for the worst case problem. (The reduction will use a polynomial number of oracle calls to A.)

The Reduction

We will assume for the reduction that we can compute permanents modulo an arbitrary prime p rather than a random prime; this isn't actually necessary but it makes the proof cleaner.

In general, if f is a degree d polynomial in n variables then you can reduce computing $f(x)$ in the worst case to computing $f(y_1), \ldots f(y_{d+1})$ for uniformly random (but not independent) y_i. The fact that the y_i are dependent will not turn out to be a serious issue: this is because we can query A several times and its output will not be history dependent. Thus if A works for most inputs, it works for most of sets of $d + 1$ inputs if each input individually looks random.

We have an algorithm A which works on most inputs; let S be the set of inputs x such that $f(x) \neq A(x)$. We are given an input x, potentially in S. The idea is that for most lines in space, most of the points on that line are not in S (because S is small). Thus if we choose a random line through x, most of the points on the line are probably not in S. Therefore we should be able to choose $d+1$ points randomly and use interpolation to find $f(x)$.

More precisely: Choose $y \in \mathbb{Z}_p^n$ uniformly at random. We will query $A(x+iy)$ for $i = 1, \ldots d+1$ and use this to indirectly find $f(x)$. Note that while $x + iy$ are not independent, each one is uniformly random. Write $h(i) = f(x + iy)$.

We know

$$\Pr_z [A(z) = f(z)] \geq 1 - \delta$$

Thus for each i,

$$\Pr_y [A(x + iy) = h(i)] \geq 1 - \delta$$

Thus by the union bound,

$$\Pr_y [\forall i : A(x + iy) = h(i)] \geq 1 - n\delta \geq \frac{2}{3}$$

If in fact $\forall i : A(x+iy) = h(i)$, then after evaluating $A(x+iy)$ at $i = 1, \ldots, d+1$ we know the value of h at $1, \ldots, d+1$. Given $d+1$ distinct points (x_i, α_i) on a degree d polynomial $t \mapsto \sum_i c_i t^i$, we can form the system of linear constraints:

$$\begin{pmatrix} x_0^0 & x_0^1 & \ldots & x_0^d \\ x_1^0 & x_1^1 & \ldots & x_1^d \\ \vdots & & & \vdots \\ x_d^0 & x_d^1 & \ldots & x_d^d \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_d \end{pmatrix} = \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_d \end{pmatrix}$$

It is a standard theorem from linear algebra that this coefficient matrix is invertible for distinct x_i. Thus there is exactly one polynomial passing through these points, and we can efficiently compute its coefficients with linear algebra. This implies in particular that $c_0 = h(0) = f(x)$. This value is correct with probability $2/3$ over the choice of y–ie over the coin flips of our algorithm. We have a BPP algorithm for evaluating $f(x)$, as desired.

3 Towards #P $\subset$ IP

We have seen that the worst-case and average-case complexity of permanent are equal. In this case we say that the permanent is random self reducible. If a function f can be computed in polynomial time on instances of size n with using an oracle for smaller instances, we say that f is downward self reducible.

Definition 2 *A function f is checkable if there is an oracle machine M which on input x with oracle A either concludes that $A(x) = f(x)$ with high probability or else finds a short proof that $\exists y : A(y) \neq f(y)$.*

Theorem 3 (Blum, Luby, Rubinfield 90) *Whenever f is random self reducible and downward self reducible then f is checkable.*

Putting this together with what we have seen so far:

Claim 4 (Nisan 90) *The permanent is checkable.*

Proof We can give an explicit algorithm for checking the permanent. We will do it by induction on the dimension of the matrix. Suppose we have verified that A correctly computes the permanent of all but $1/k$ of the matrices of size k. We will inductively verify that A correctly computes the determinants of all but $1/(k+1)$ of the matrices of size $k+1$. To do this, pick a random $k+1 \times k+1$ matrix M and query $A(M)$. Recall from last class that we can write $\text{Perm}(M)$ as a linear combination of the permanents of $k+1$ $k \times k$ matrices $N_1, \ldots N_{k+1}$. (Expand by minors, splitting up the sum of $\sum_\sigma \prod_i M_{i\sigma(i)}$ based on the value $\sigma(1)$.) Given that A correctly computes the permanents of most $k \times k$ matrices, we can use the random self reduction for permanent to compute the permanent of each N_i with very high probability. Then we can combine these results to learn with very high probability whether A correctly computed the permanent of M. If we repeat this experiment for many randomly chosen M, we can determine whether A correctly computes permanents of $k+1 \times k+1$ matrices.

Once we know that A correctly computes permanents of most matrices of dimension $\dim(M)$, we can use our random self reduction to determine whether it correctly computes the determinant of any particular matrix M.

■

The notion of checkability is related to an interactive proof, but not the same. The difference is that the protocol A which is being checked responds in a fixed way to each possible query. A prover on the other hand may adjust their responses based on the history of their interaction with the verifier. This may allow a prover to fool a verifier into believing that they are correct even if no program could convince a checker that they are correct.

4 Towards PSPACE $\subset$ IP

Theorem 5 (Lunel, Fortnow, Karloff, Nisan 90) $\#\text{P} \subset \text{IP}$.

Theorem 6 (Shamir 90) $\text{PSPACE} \subset \text{IP}$.

To prove PSPACE $\subset$ IP, we will use the fact that PSPACE languages are computed in polynomial time by an alternating Turing machines. We would like to replace existential quantifiers by the prover and the universal queries by the verifier. When we come to a an existential quantifier, we just need to ask the verifier which way to go. This works fine: the prover will try and convince us of the truth of his assertion, so he will honestly report which branch accepts. Universal quantifiers are more difficult because we must somehow check both computation paths even though the prover is not motivated to help us.

We would like to verify that both computation paths accept with a single query. This new query should be false with high probability if either of the original queries is false, while it should be true if both are true. If we could do this, we could simultaneously test both branches. This is the reverse of what we wanted before: earlier we wanted to split a single query into a number of random queries whose conjunction was equivalent to the original query, where here we want to combine a number of smaller queries into a single query equivalent to their conjunction.

Suppose I have a polynomial P and I want to check its value on a number of points. The prover makes some claims about the values of P at these points. The way we will check all of these claims simultaneously is to take a curve which goes through those points, have the prover give us the values of the polynomial on that curve, and then check this purported value at a random point on the

curve. If any of the prover's original claims are wrong, then the prover will probably also be wrong about the value of P at this random point. We now make this intuitive approach slightly more precise.

Suppose that the verifier is claiming that $P(x_i) = \alpha_i$ for each $i = 1, \ldots, w$ for a polynomial P of degree d of n variables. We want to find y and β such that if $P(y) = \beta$ then $P(x_i) = \alpha_i$ for each i (and conversely).

The idea is based on the notion of a curve. A curve C is a collection of polynomials $C^{(i)}$ defining a function $C : \mathbb{Z}_p \to \mathbb{Z}_p^n$ by $t \mapsto (C^{(1)}(t), \ldots C^{(n)}(t))$. The degree of a curve C is defined to be $\max_i \deg C^{(i)}$. The notion of a curve is useful because if $f : \mathbb{Z}_p^n \to \mathbb{Z}_p$ is a polynomial, then $f \circ C$ is a polynomial of 1 variable of degree $\deg C \deg f$.

Given w points x_i in space, we can find a degree $w - 1$ curve C such that $C(i) = x_i$ for each i by doing normal polynomial interpolation on each of the n coordinates.

Now returning to our points x_i for which the prover claims $P(x_i) = \alpha_i$; pick a curve C such that $C(i) = x_i$ for each $i \leq w$. The prover has already made a claim about the values $P(C(1)), P(C(2)) \ldots P(C(w))$. The verifier sends C to the prover, who replies with a function h which the prover claims is equal to $f \circ C$ (since this is a low-degree univariate polynomial, the prover can send its description). The verifier can now check that h is consistent with the prover's claims for values of $f \circ C$ at $1, 2, \ldots w$. If not, then the prover has already lied, so the verifier can reject. Otherwise, the verifier can pick a random $t \in \mathbb{Z}_p$ and attempt to verify that $f(C(t)) = h(t)$. If the prover is honest, this is easy to do. If the prover is attempting to lie about one of the values $C(i) = x_i$ then the prover is forced to lie about h. But if the prover lies about h, then $h(x) - f(C(x))$ is a non-zero polynomial and is therefore non-zero for most x. Thus if the prover is lying, it is probably the case that $f(C(t)) \neq h(t)$. But this is precisely what we wanted: the prover will be forced to commit to the single false claim $f(C(t)) = h(t)$ unless $f(C(i)) = \alpha_i$ for each i.

5 Finishing #P $\subset$ IP

We can now give a reduction from permanent for $k + 1 \times k + 1$ matrices to permanent for $k \times k$ matrices which works in the context of an interactive proof instead of a program checker; we will do this by combining the random reduction for permanent with the downward reduction for permanent, using the ideas above. We have some matrix M, and we would like to verify the prover's claim that $\mathrm{Perm}(M) = \alpha$. We can write $\mathrm{Perm}(M) = \sum_i M_{1i} \mathrm{Perm}(N_i)$, where N_i are the minors of M. Next we can construct a curve C in $\mathbb{Z}_p^{k^2}$ which passes through every N_i. Now ask the prover to provide $h(x) = \mathrm{Perm}(C(x))$. Supposing h is correct, we can use it to compute $\mathrm{Perm}(M)$: we take an appropriate linear combination of $h(1), h(2), \ldots h(k+1)$. In order to verify h, we choose a random $t \in \mathbb{Z}_p$ and inductively verify that $h(t)$ is really the permanent of $C(t)$.

Clearly if the prover is honest they will convince the verifier with probability 1. Suppose inductively that for every matrix the prover convinces the verifier with probability s_k on $k \times k$ matrices (over the coin flips of the verifier). Then if the prover attempts to lie by providing an $h(x) \neq \mathrm{Perm}(C(x))$, with probability $1 - \frac{k+1}{p}$ over the random choice of t, the verifier will choose a t such that $h(t) \neq \mathrm{Perm}(C(t))$ (since non-zero univariate degree $k + 1$ polynomials have at most $k + 1$ roots). Then the prover convinces the verifier with probability $s_{k+1} \leq n/p + s_k$. Since the permanent of 1×1 matrices is easily computed, $s_1 = 0$. If p is an n-bit prime this implies that $s_n = \mathcal{O}(n^2/p)$ is negligible. This

is just what we wanted.