CS 810: Complexity Theory	3/28/2007

Lecture 24: Probabilistically Checkable Proofs

Instructor: Dieter van Melkebeek — Scribe: Tom Watson

In this lecture we continue our discussion of interactive proof systems. We develop a nonrelativizing proof technique, encapsulated in the so-called *sumcheck protocol*, and use it to prove that $\mathrm{P}^{\#\mathrm{P}} \subseteq \mathrm{IP}$. Building on this technique, we strengthen the result and show that IP = PSPACE, i.e., polynomial space exactly captures the power of interactive proof systems. Then we introduce the notion of *multiple prover interactive proof systems* and relate them to *probabilistically checkable proofs*, in which the interaction between prover and verifier is eliminated but the proof is allowed to be very long. There are some extremely profound and involved results in these areas, which we briefly discuss.

1 Interactive Proof Systems for #P

Two lectures ago we discussed proof complexity, in which the central question is whether coNP statements have short proofs. In the last lecture we generalized our notion of proof to include both randomness in the verification and interaction with a prover. This led to the notion of an *interactive proof system*, in which there are two parties interacting to decide whether a given input is in a particular language. There is a *prover* who is computationally unrestricted, and therefore knows whether the input is in the language, but it is the (skeptical) probabilistic *verifier* who gets the final say. We can think of the prover as trying to get the verifier to accept, and we want it to be the case that if the input is in the language, then some prover can, in fact, convince the verifier to accept with probability at least some value c (informally, we can prove everything that's true, the *completeness* property) and that if the input is not in the language, then *no* prover, not even a devious one, can convince the verifier to accept with probability greater than some value s (informally, we cannot prove anything that's false, the *soundness* property).

We gave examples of interactive proof systems for the Graph Nonisomorphism problem, which is a problem for which we don't know of short, efficiently verifiable classical proofs (although under certain reasonable complexity theoretic assumptions, they are known to exist). We now demonstrate that not only do all coNP statements have short proofs in the present sense, but (presumably) many more languages have short proofs. We establish the exact power of interactive proof systems by showing that IP = PSPACE. We point out that the proof is nonrelativizing. As a step toward proving this result, we now develop an interactive proof for the problem of deciding whether the number of satisfying assignments to a given CNF formula is exactly some number k. This allows us to establish that $\mathrm{P}^{\#\mathrm{P}} \subseteq \mathrm{IP}$.

Our interactive proof system involves two main ingredients. The first is arithmetization of the given formula ϕ, as discussed in the last lecture. This is a simple procedure that produces a multivariate polynomial $\widetilde{\phi}(x_1, \ldots, x_n)$ that agrees with ϕ on the Boolean cube. This is the step that prevents our proof from relativizing; we leave it as an exercise to argue formally why this is the case. This polynomial has two very important properties: every variable has degree at most m in it, where m is the number of clauses in ϕ, and it can be efficiently evaluated (despite possibly having exponentially many monomials when expanded). These are the only two properties we need in order to apply the second main ingredient, the sumcheck protocol, which we describe next.

1.1 Sumcheck Protocol

Our goal is to design a verifier to decide whether the number of satisfying assignments to ϕ is equal to some number k_0. This is equivalent to checking the following identity:

$$\sum_{x_1=0}^{1}\sum_{x_2=0}^{1}\cdots\sum_{x_n=0}^{1}\widetilde{\phi}(x_1,x_2,\ldots,x_n)=k_0. \tag{1}$$

We don't have time to evaluate $\widetilde{\phi}$ for all 2^n settings of $x_1,\ldots,x_n$, so we enlist the help of the prover. The sumcheck protocol is a method for doing this that is secure against cheating provers.

The key idea is to consider the univariate polynomial $g(x)$ defined by

$$g(x)=\sum_{x_2=0}^{1}\cdots\sum_{x_n=0}^{1}\widetilde{\phi}(x,x_2,\ldots,x_n).$$

This is the same as the expression in Equation (1) except that we don't sum over x_1. Since x_1 has degree at most m in $\widetilde{\phi}$, $g(x)$ has degree at most m. If we knew $g(x)$, then we would be able to check Equation (1) by checking $g(0)+g(1)=k_0$, using the observation that

$$g(0)+g(1)=\sum_{x_1=0}^{1}\sum_{x_2=0}^{1}\cdots\sum_{x_n=0}^{1}\widetilde{\phi}(x_1,x_2,\ldots,x_n).$$

We don't have $g(x)$, but we can ask the prover to send it to us. We know that $g(x)$ has at most $m+1$ coefficients, which is manageable, but it's conceivable that these coefficients are prohibitively huge numbers. This turns out not to be an issue, but we postpone the discussion of this.

Now the prover sends us some polynomial $g'(x)$, which may not be equal to $g(x)$ if the prover is cheating. We check that $g'(x)$ has degree at most m and that $g'(0)+g'(1)=k_0$ and reject if either does not hold. Now if Equation (1) is true, then the prover can just send us the correct polynomial $g'(x)=g(x)$ and everything will be fine. If Equation (1) is not true, then if the prover wants to have a prayer of getting us to accept, then it's forced to send an incorrect polynomial $g'(x)\neq g(x)$. Since the two polynomials agree in at most m places, we can pick a random number ξ_1 from a reasonably small set I and with high probability, $g'(\xi_1)\neq g(\xi_1)$. If we had $g(x)$ then we could just evaluate $g(\xi_1)$ and $g'(\xi_1)$ and with high probability catch the cheating prover. Of course, the whole point is that we don't have $g(x)$, but we have now reduced our original problem, verifying Equation (1), to a simpler instance of the same problem, namely verifying

$$\sum_{x_2=0}^{1}\cdots\sum_{x_n=0}^{1}\widetilde{\phi}(\xi_1,x_2,\ldots,x_n)=k_1 \tag{2}$$

where $k_1=g'(\xi_1)$. By sending us $g'(x)$, the prover has implicitly claimed that Equation (2) holds. It had no way of knowing what number ξ_1 we would pick, though, which gives us an edge. If Equation (1) holds, then Equation (2) holds when the prover just sends us the correct polynomial. If Equation (1) does not hold, then with high probability Equation (2) does not hold, regardless of the prover's behavior. In either case we are back to where we started, except that now there are only $n-1$ variables being summed over. We iterate this process until our task is reduced to the problem of verifying

$$\widetilde{\phi}(\xi_1,\xi_2,\ldots,\xi_n)=k_n \tag{3}$$

for some numbers $\xi_1, \ldots, \xi_n \in I$ and k_n. Now we use the other property required of $\widetilde{\phi}$, namely that is can be evaluated efficiently, to explicitly check that Equation (3) holds. We accept if it holds and reject otherwise. This is the only time in the protocol when we are willing to accept.

Theorem 1. *The sumcheck protocol is a public-coin interactive proof system that verifies Equation (1) with perfect completeness and polynomially small soundness, provided $\widetilde{\phi}$ has degree at most m in every variable and can be evaluated in polynomial time.*

Proof. That the protocol is public-coin is clear — in each step the verifier uses its randomness only to pick the number ξ_i, and then it sends ξ_i to the prover. That the protocol has perfect completeness is also straightforward to see — if Equation (1) holds then the prover can just send us the correct polynomial $g'(x) = g(x)$ in every phase and the consistency check $g'(0) + g'(1) = k_i$ will always pass and all the implicit claims of the form of Equation (2) will be true, so the final check of Equation (3) will also pass, leading to acceptance with probability 1.

Now we argue the soundness. Suppose Equation (1) does not hold, but nevertheless we accept. In the last phase, the claim we're checking, Equation (3), is true (otherwise we would reject), but in the first phase, the claim we're checking, Equation (1), is false. Thus there must be some phase where the current claim transitions from being false to being true. In this phase, the prover sends us some $g'(x) \neq g(x)$ (since otherwise we would reject after seeing that the consistency check $g'(0) + g'(1) = k_i$ fails). For the claim generated by this phase to be true, it must be the case that $g'(\xi_{i+1}) = g(\xi_{i+1})$, which happens with probability at most $m/|I|$. The probability that we accept is at most the probability that this happens in some phase, which by a union bound is at most $mn/|I|$. We can choose $|I|$ to be polynomially large to make the soundness polynomially small.

The only thing left to verify is that the protocol runs in polynomial time. The final check can be done in polynomial time by hypothesis, so the only issue is that the coefficients of the $g(x)$ polynomials might become too large. One can argue that the coefficients do not become too large. We can avoid that analysis by letting I be the integers mod p for a small prime p, and letting all computations take place in the field of integers mod p. Now we need to worry about the soundness, though; it could be the case that the originally claimed value k_0 is congruent to the true value mod p, and in that case the prover can make the verifier always accept. However, the difference between k_0 and the true value cannot have too many prime factors (certainly fewer than n). The density of primes is great enough that picking p at random from the set of primes of polynomial magnitude (and hence logarithmic bit length) affects the soundness by only a polynomially small amount. Thus the overall soundness is still polynomially small, and computations mod p can certainly be done efficiently. Selecting p at random is not a problem since the verifier can enumerate all primes in the desired range by brute force. □

Corollary 1. *Every language in* $\mathrm{P}^{\#\mathrm{P}}$ *has a public-coin interactive proof system with perfect completeness, and in particular* $\mathrm{P}^{\#\mathrm{P}} \subseteq \mathrm{IP}$.

Proof. We can simulate a $\mathrm{P}^{\#\mathrm{P}}$ machine, and whenever it makes an oracle query, reduce the query to a #SAT instance, have our prover tell us how many satisfying assignments it has, and run the sumcheck protocol to verify this. The resulting protocol still runs in polynomial time. If the input is in the language, then the prover can follow the protocol, leading to acceptance with probability 1. If the input is not in the language, then for us to accept, the prover must cheat on at least one query, and this slips by us with exponentially small probability per query. By a union bound, the soundness of this protocol is still exponentially small. □

2 Interactive Proof Systems for PSPACE

Using the sumcheck protocol, we can now characterize the power of interactive proof systems.

Theorem 2. *Every language in* PSPACE *has a public-coin interactive proof system with perfect completeness, and in particular* PSPACE $\subseteq$ IP*.*

Corollary 2. IP = PSPACE.

Proof. The inclusion PSPACE $\subseteq$ IP follows from Theorem 2. The inclusion IP $\subseteq$ PSPACE is left as an exercise; it just involves a space-efficient simulation of an interactive proof system. □

In the last lecture we mentioned that every interactive proof system can be converted to one with perfect completeness without increasing the number of rounds, and also to one with both perfect completeness and public coins with only an increase of two in the number of rounds. In particular, requiring perfect completeness and public coins does not decrease the power of IP. Theorem 2 and Corollary 2 provide an alternative proof of the latter fact: every language in IP is in PSPACE by Corollary 2, and every language in PSPACE has a public-coin interactive proof system with perfect completeness by Theorem 2.

We now prove the main result of this section.

Proof of Theorem 2. One idea is to show that $TQBF \in$ IP by arithmetizing quantified formulas and using a protocol similar to the sumcheck protocol. Arithmetizing a universal quantifier, however, results in a squaring of the degree, which could lead to very high degree polynomials. This can be remedied by using an extra step that reduces the degree of the polynomial after arithmetizing a universal quantifier. This approach does lead to a proof that PSPACE $\subseteq$ IP, but we go a different route, employing the divide-and-conquer strategy used in our proof that NSPACE(s) $\subseteq$ DSPACE(s^2).

In the sumcheck protocol, our goal was to count the number of satisfying assignments to a CNF formula, or equivalently the number of accepting computation paths of a nondeterministic machine. Now we are given a deterministic machine running in space s and wish to determine the number of accepting computation paths, which is either 0 or 1. The computation paths, however, could be exponentially long, so we can't work directly with a formula expressing the whole computation. We instead work with polynomials derived from the divide-and-conquer approach of verifying a computation tableau. This involves another application of the main idea of the sumcheck protocol — in a divide-and-conquer step, we do not just randomly pick a half of the tableau to recurse on, because this only gives us a probability 1/2 of catching a cheating prover. Instead we capture this with a polynomial where plugging in 0 corresponds to checking one half of the tableau and plugging in 1 corresponds to checking the other half, and we plug in a random value from a much larger domain. In this way we sort of "handle both halves at once", and catch a cheating prover with high probability.

We now pursue this intuition formally. We assume for simplicity of notation that the configurations of the machine M we wish to simulate are bit strings of length s, and that the machine makes exactly 2^s transitions, where 2^s is a power of 2 (thus the computation tableau has 2^s+1 rows). This last assumption can be justified by assuming that once M reaches a halting configuration, it just keeps transitioning to that same configuration forever. We also assume there is a unique accepting configuration. For $\ell = 0, \ldots, s$ we define the following predicate on pairs of configurations:

$$P_\ell(C_0, C_1) = \begin{cases} 1 & \text{if } C_0 \vdash_M^{2^\ell} C_1 \\ 0 & \text{otherwise} \end{cases}.$$

That is, $P_\ell(C_0, C_1)$ indicates whether executing 2^ℓ transitions of M starting from configuration C_0 leads to configuration C_1. These predicates satisfy the following recurrence: P_0 is just specified by the transition function of M, and for $\ell > 0$,

$$P_\ell(C_0, C_1) = \sum_{C_{1/2} \in \{0,1\}^s} P_{\ell-1}(C_0, C_{1/2}) P_{\ell-1}(C_{1/2}, C_1). \tag{4}$$

Thus, the claim that we need to verify is

$$P_s(C_0, C_1) = k_s, \tag{5}$$

where C_0 is the unique initial configuration determined by the input, C_1 is the unique accepting configuration, and $k_s = 1$.

Our first step is to arithmetize these predicates to obtain multivariate polynomials (in fact, polynomials in exactly $2s$ variables) $\widetilde{P}_\ell(C_0, C_1)$ such that $\widetilde{P}_\ell$ agrees with P_ℓ on the Boolean cube. We have the following fact.

Claim 1. *Under an appropriate encoding of configurations, there exists a $2s$-variate polynomial $\widetilde{P}_0(C_0, C_1)$ that agrees with $P_0(C_0, C_1)$ on the Boolean cube, can be evaluated in polynomial time, and such that the degree of every variable is $O(1)$.*

We leave the proof of Claim 1 as an exercise. It just involves summing over all possible tape head positions and transitions, and expressing that the contents of the configurations are consistent with that transition by multiplying together many simple checks.

The polynomial $\widetilde{P}_0$ and Equation (4) immediately specify all the polynomials $\widetilde{P}_\ell$. Moreover, in each $\widetilde{P}_\ell$, every variable has degree $O(1)$. This can be argued by induction, using the fact that in Equation (4), the only variables whose degrees increase are $C_{1/2}$, but these are summed out. The constant degree property is not critical, though; it turns out that our argument would go through if the degrees of the variables were larger.

The claim we need to verify is now

$$\widetilde{P}_s(C_0, C_1) = k_s. \tag{6}$$

Now looking at Equation (4), we see that if $\widetilde{P}_{s-1}$ were easy to evaluate, then we would be in exactly a position to use the sumcheck protocol to verify our claim and be done. We don't know how to evaluate $\widetilde{P}_{s-1}$ efficiently, but we observe that actually, the sumcheck protocol reduces the problem of verifying

$$\sum_{C_{1/2} \in \{0,1\}^s} \widetilde{P}_{s-1}(C_0, C_{1/2}) \widetilde{P}_{s-1}(C_{1/2}, C_1) = k_s$$

to the problem of verifying

$$\widetilde{P}_{s-1}(C_0, \gamma) \widetilde{P}_{s-1}(\gamma, C_1) = \kappa_s \tag{7}$$

for some particular values γ and κ_s.

It looks as if we've almost reduced our original problem, checking Equation (6), to a simpler instance of the same problem. However, Equation (7) involves the product of two evaluations of $\widetilde{P}_{s-1}$, so it is not quite the same as our original problem. The two factors correspond to the two halves of the computation tableau, and if we were to recurse on both halves, then we would be no better off than just simulating the entire computation from start to finish. The following is the key idea that leads to an exponential savings in running time.

Define $h(x)$ to be a univariate polynomial such that $h(0) = (C_0, \gamma)$ and $h(1) = (\gamma, C_1)$. More precisely, h is a $2s$-tuple of univariate polynomials, one for each bit of the output. Each bit is a simple linear function of x (since C_0, γ, and C_1 are fixed). Now we're interested in checking

$$\widetilde{P}_{s-1}(h(0))\widetilde{P}_{s-1}(h(1)) = \kappa_s,$$

which seems similar to the consistency check $g'(0) + g'(1) = k_0$ in the sumcheck protocol. Here we set $g(x) = \widetilde{P}_{s-1}(h(x))$. Since each of the $2s$ components of h has degree at most 1, and each of the $2s$ variables in $\widetilde{P}_{s-1}$ has degree $O(1)$, $g(x)$ is a univariate polynomial of degree $O(s)$. If we had $g(x)$ then we could just check that $g(0)g(1) = \kappa_s$ and be done. Like in the sumcheck protocol, we ask the prover for the coefficients of $g(x)$ and it gives us some $g'(x)$ of degree $O(s)$. We check that $g'(0)g'(1) = \kappa_s$ and reject if this does not hold. In the event that $g(0)g(1) \neq \kappa_s$, this forces the prover to give us an incorrect polynomial $g'(x) \neq g(x)$. In order to catch a cheating prover, we pick a random $\xi \in I$, and now with high probability, $g'(\xi) \neq g(\xi)$. We can't detect this immediately since we can't evaluate $g(\xi)$, but we have reduced our original problem to a simpler instance of the same problem. By sending $g'(x)$, the prover has implicitly claimed that

$$\widetilde{P}_{s-1}(h(\xi)) = k_{s-1}, \tag{8}$$

where $k_{s-1} = g'(\xi)$. Now k_s has been replaced by k_{s-1}, (C_0, C_1) has been replaced by $h(\xi)$, and ℓ has been reduced from s to $s-1$. We can iterate this process until we've reduced the problem to checking

$$\widetilde{P}_0(\alpha) = k_0, \tag{9}$$

where α is some $2s$-tuple of numbers (not necessarily bits) and k_0 is some number. By Claim 1, this can be done efficiently. We accept if Equation (9) holds and reject otherwise. This is the only time in the protocol when we are willing to accept.

As in the case of the sumcheck protocol, we need to worry about whether the coefficients become too large. In this case, they can become very large, which necessitates performing all computations modulo some small prime. Since the number of accepting computation paths is either 0 or 1, the soundness is unaffected, regardless of which prime we use.

We now recap the protocol and argue its correctness. The main outline consists of s iterations, each of which reduces the problem of verifying the value of $\widetilde{P}_\ell$ on some input to the problem of verifying the value of $\widetilde{P}_{\ell-1}$ on some other input, and then a final verification of the value of $\widetilde{P}_0$ on some input, which can be done efficiently. Each iteration consists of running the sumcheck protocol with the identity (4) to reduce the problem of verifying the value of $\widetilde{P}_\ell$ on some input to the problem of verifying the value of the product of $\widetilde{P}_{\ell-1}$ on two other inputs. This is then reduced to the problem of verifying the value of $\widetilde{P}_{\ell-1}$ on a single input by interpolating through the two inputs with a univariate polynomial $h(x)$ and setting the single input to be $h(\xi)$ for a randomly chosen ξ.

Like the sumcheck protocol, this protocol is public-coin and has perfect completeness. Now if the original claim, Equation (6), is false, then in order for us to accept, the current claim must transition from being false to being true in some iteration. By a union bound, the soundness is at most s times the probability that this happens in any given iteration. Within an iteration, there are two steps: we obtain an intermediate claim via the sumcheck protocol, and then we obtain the claim for the next iteration using the interpolating polynomial $h(x)$ evaluated at a random ξ. The probability that the intermediate claim becomes true is at most $O(s/|I|)$ by Theorem 1 (since there are $O(s)$ variables and each has constant degree). The probability that the claim transitions from false to true in the second step is at most $O(s/|I|)$ since the $g(x)$ has degree $O(s)$. By a union bound, the probability that the claim transitions from false to true in a given iteration is also $O(s/|I|)$, and so the soundness of the protocol is $O(s^2/|I|)$. Since s is only polynomial, $|I|$ doesn't have to be too large to make the soundness small, and thus we don't have to pick the prime p to be very large. We can find an appropriate p by brute force. $\square$

3 Probabilistically Checkable Proofs

3.1 Multiple Prover Interactive Proof Systems

We bridge the gap between interactive proof systems and probabilistically checkable proofs by introducing *multiple prover interactive proof systems*, in which there are multiple provers that the verifier may query throughout the protocol, and they are allowed to cooperatively decide on any strategy before the protocol begins, *but they may not communicate during the protocol*. A multiple prover interactive proof system is said to decide a language L with completeness c and soundness s if for all $x \in L$ there exist provers that make the verifier accept with probability at least c, and for all $x \notin L$, no provers can make the verifier accept with probability more than s. Define the class MIP to be the set of languages decided by polynomial-time multiple prover interactive proof systems with any number of provers (though the verifier can clearly only query polynomially many) and say completeness 2/3 and soundness 1/3.

The additional power afforded by having multiple provers is similar to a strategy used by the police to question suspected accomplices: interrogating the suspects separately and checking the consistency of their stories improves the chances of being able to catch them lying. Just as we quantified the exact power of interactive proof systems in Corollary 2, one can quantify the exact power of multiple prover interactive proof systems as follows.

Theorem 3. MIP = NEXP.

We omit the proof of Theorem 3, which uses similar ingredients as our proof of Theorem 2 but is more involved. We note, however, that this theorem and the fact that NP $\subsetneq$ NEXP (which follows from the nondeterministic time hierarchy) imply that we can decide *provably* more languages with multiple prover interactive proof systems than with classical proofs (i.e., NP).

Since we believe that PSPACE $\subsetneq$ NEXP, it seems that multiple prover interactive proof systems are strictly more powerful than single prover interactive proof systems. Does each additional prover yield more power? The following theorem, which we prove shortly, shows that the answer is no. Define MIP$[k]$ to be the set of languages decided by multiple prover interactive proof systems with k provers.

Theorem 4. MIP = MIP[2].

We are now ready to study the very influential notion of probabilistically checkable proofs, in which the proof is entirely written down and the element of interaction is removed, but the proofs are in principle allowed to be very long.

3.2 Probabilistically Checkable Proofs

Definition 1. *A* probabilistically checkable proof system *with completeness c and soundness $s < c$ for a language L is a polynomial-time probabilistic oracle Turing machine V such that if $x \in L$ then there exists a proof Π such that*

$$\Pr[V^{\Pi}(x) \text{ accepts}] \geq c,$$

and if $x \notin L$ then for all proofs Π,

$$\Pr[V^{\Pi}(x) \text{ accepts}] \leq s.$$

Typical settings of the completeness and soundness parameters are $c = 1$, $s = 1/2$.

The key difference between a probabilistically checkable proof system and an interactive proof system is that in the former, the prover is a fixed oracle — its responses to all possible queries must be fixed for a given input and are not allowed to change after the verifier starts computing. With probabilistically checkable proofs, we are usually interested in the amount of randomness and number of queries needed by the verifier. This motivates the following definition.

Definition 2. *The class* $\mathrm{PCP}(r(n), q(n))$ *is the set of languages decidable by a probabilistically checkable proof system where the verifier uses at most $r(n)$ random bits and queries at most $q(n)$ bits of the proof.*

Theorem 5. $\mathrm{PCP}(r(n), q(n)) \subseteq \mathrm{NTIME}(2^{O(r(n)+q(n))} n^{O(1)})$.

Proof. Consider a probabilistically checkable proof system where the verifier uses $r(n)$ random bits and makes $q(n)$ queries. For each random bit sequence, the number of locations of the proof that could ever get queried is $2^{q(n)}$ (due to the possibly adaptive nature of the verifier). Thus over all proofs and all random strings, there are at most $2^{r(n)+q(n)}$ locations that ever get queried (for a given input). A nondeterministic machine can guess the addresses and answers to these queries in time $2^{r(n)+q(n)} n^{O(1)}$. It can then run over all random bit sequences, simulate the verifier for each one (looking up the answers to queries in the guessed table), and explicitly compute the probability of acceptance of the verifier for the guessed proof. This takes time $2^{r(n)} n^{O(1)}$. If the probability of acceptance is at least the completeness of the probabilistically checkable proof system then the machine accepts, and otherwise it rejects. □

We now draw the connection between multiple prover interactive proof systems and probabilistically checkable proofs. Note that the following result immediately implies Theorem 4 and shows that $\mathrm{MIP} = \mathrm{PCP}(poly(n), poly(n))$.

Theorem 6. $\mathrm{MIP} \subseteq \mathrm{PCP}(poly(n), poly(n)) \subseteq \mathrm{MIP}[2]$.

Proof. We first argue the inclusion $\mathrm{MIP} \subseteq \mathrm{PCP}(poly(n), poly(n))$. For a given multiple prover interactive proof system, we construct a probabilistically checkable proof system where the proof is interpreted as a complete specification of the provers' strategies, i.e., a list of their responses for every possible message history. The verifier can just simulate the multiple prover interactive proof system, looking up the provers' responses by querying the provided proof. The completeness and soundness are the same as for the multiple prover interactive proof system.

For the inclusion $\mathrm{PCP}(poly(n), poly(n)) \subseteq \mathrm{MIP}[2]$, we can have the verifier just simulate the given probabilistically checkable proof system, using one of its provers to answer queries to the proof. The only problem is that this prover is allowed to be adaptive, whereas in the probabilistically checkable proof system, the entire proof is written down from the beginning. We use a second prover to try to catch the first prover if it behaves nonadaptively. However, the second prover is also allowed to be adaptive, so we're only safe asking it one question. Specifically, when the simulation is over, if the verifier decides to accept then before doing so it picks one of the queries it made to the first prover uniformly at random, makes the same query to the second prover, and only proceeds to accept if their answers agree. Since the provers cannot communicate during the protocol, the second prover's possible responses form a nonadaptive strategy.

Now if the input is in the language, then the two provers can just respond according to the proof specified by the probabilistically checkable proof system, leading to acceptance to probability at least the completeness of that system. If the input is not in the language, then we can compute the probability of acceptance as follows, where q is the number of queries made and A denotes the event that for at least one query asked of the first prover, its response differs from the response we would get if we asked the second prover.

$$\begin{aligned}
\Pr\left(\text{accept}\right) &= \Pr\left(\text{accept} \;\middle|\; A\right) \cdot \Pr\left(A\right) + \Pr\left(\text{accept} \;\middle|\; \overline{A}\right) \cdot \Pr\left(\overline{A}\right) \\
&\le \Pr\left(\text{accept} \;\middle|\; A\right) + \Pr\left(\text{accept} \;\middle|\; \overline{A}\right) \\
&\le \left(1 - \frac{1}{q}\right) + s.
\end{aligned}$$

In the last inequality, $\Pr\left(\text{accept} \;\middle|\; A\right) \le 1 - 1/q$ follows from the fact that we catch the provers' inconsistency with probability at least $1/q$, and $\Pr\left(\text{accept} \;\middle|\; \overline{A}\right) \le s$ follows from the soundness property applied to the proof defined by the second prover's responses.

Now the original probabilistically checkable proof system can be amplified so that s is exponentially small and c is exponentially close to 1. Then since q is only polynomially large, the soundness $1 - 1/q + s$ of this multiple prover interactive proof system is non-negligibly less than its completeness c. We can repeat the whole protocol a few times and see whether the average number of accepted runs is less than or at least the midpoint between $1 - 1/q + s$ and c. Intuitively, there shouldn't be issues with adaptivity when we rerun the protocol from scratch since the queries made in previous runs give the provers no information about what will happen in future rounds. More formally, each possible execution of the first $k-1$ runs defines a proof for the original probabilistically checkable proof system, namely, the proof specified by the responses of the second prover in the kth run given the execution of the first $k-1$ runs. The soundness and completeness properties apply to each of these exponentially many proofs corresponding to the different executions of the first $k-1$ runs. Now suppose the input is not in the language. Then the probability of acceptance on the kth run given any particular execution of the first $k-1$ runs is at most $1 - 1/q + s$ and hence the random variable indicating acceptance on the kth run is at most some indicator random variable that is 1 with probability exactly $1 - 1/q + s$ conditioned on each execution of the first $k-1$ runs. It follows that the indicator random variables thus defined for each of the runs are fully independent and all have expectation $1 - 1/q + s$, so by a Chernoff bound the soundness is driven down exponentially. Similarly, the completeness is improved. $\square$

3.3 The PCP Theorem

By the nondeterministic time hierarchy theorem, we know that there exist languages in NEXP that do not have short, efficiently verifiable proofs. However, Theorems 3 and 6 imply that all languages in NEXP have efficiently verifiable proofs, where we allow the verifier to randomly spot check a few locations of the proof. Can we scale this result down to get a similar result for NP? That is, can we replace classical proofs with ones where it is only necessary to spot check a few random locations? This is the content of the famous PCP Theorem, which is possibly the most celebrated and involved result in all of theory of computing.

Theorem 7 (The PCP Theorem). $\mathrm{NP} = \mathrm{PCP}(O(\log n), O(1))$.

The inclusion $\mathrm{PCP}(O(\log n), O(1)) \subseteq \mathrm{NP}$ follows immediately from Theorem 5. The other direction is much harder. We do not prove this result in this course, but we illustrate the flavor of the proof by proving the following weaker result, which uses some of the same techniques needed for the PCP Theorem.

Theorem 8. $\mathrm{NP} \subseteq \mathrm{PCP}(\mathit{poly}(n), O(1))$.

In the probabilistically checkable proof system we design for Theorem 8, the proof is an exponentially long encoding of a classical proof. Nevertheless, this result captures what is perhaps the most surprising aspect of the PCP Theorem, namely that only a constant number of bits of the proof need to be queried to verify its correctness.

One of the major applications of the PCP Theorem is hardness of approximation results. Many important combinatorial optimization problems are NP-hard to solve exactly, and for many of these problems the PCP Theorem implies that it is NP-hard even to approximate them within certain factors. In fact, for some problems we have tight results, i.e., approximation algorithms and hardness results with essentially matching approximation factors. Thus the PCP Theorem has allowed us to characterize the approximability of these problems. The basic reason the PCP Theorem leads to hardness of approximation results is that the prover is trying to solve an optimization problem — that of maximizing the probability that the verifier accepts — and this theorem introduces a non-negligible gap in this probability between the cases where the input is in the language and not in the language.

In the next lecture, we will see concrete examples of how the PCP Theorem leads to hardness of approximation results. We will also prove Theorem 8. The proof involves encoding classical proofs with the Hadamard code and using harmonic analysis to analyze the properties of the resulting probabilistically checkable proof system.

4 The Power of the Honest Prover

In this section we address the following question and its implications. For a given interactive or probabilistically checkable proof system, if the input is in the language, then how powerful does a prover have to be to convince the verifier of this fact? We would like it to be the case that the honest prover isn't required to perform computational tasks that are more difficult than solving the original problem from scratch. That is, ideally, the honest prover can answer queries in polynomial time with access to an oracle for the language being decided. We have seen several examples where this is the case.

- This is the case for our private-coin interactive proof system for Graph Nonisomorphism: in the case that the two input graphs are non-isomorphic, all the prover has to do is determine which of the two is isomorphic to the graph provided by the verifier, which can be done with two queries to an oracle for Graph Nonisomorphism.
- This is also the case for our sumcheck protocol for #P functions. The prover is only ever required to find the coefficients of a univariate polynomial obtained by summing an easily evaluatable multivariate polynomial over all 0-1 settings of all but one variable. The prover can plug in a particular value for the free variable and use a #P oracle to find the value of the desired polynomial on that input. (The basic reason is the closure of #P under uniform exponential summations. The #P oracle just needs to nondeterministically guess the settings of the other variables, evaluate the multivariate polynomial, and generate a number of accepting branches equal to the outcome.) Since the desired polynomial is of low degree, the prover can try a few different inputs and reconstruct the polynomial from its values on those inputs.
- Finally, the prover for our PSPACE protocol can compute its responses in polynomial space. After all, the proof of the inclusion $\mathrm{IP} \subseteq \mathrm{PSPACE}$ involves simulating the interactive proof system and determining the strategy that causes the verifier to accept with the highest probability. The prover can follow this strategy.

We have also seen some examples for which it is open whether an honest prover requires more computational resources than are required to solve the problem at hand.

- This is the case for our public-coin interactive proof system for Graph Nonisomorphism.
- Theorems 1 and 2 immediately yield interactive proof systems for coNP. It is open whether an honest prover for either of these protocols can compute its responses with a coNP oracle. The most efficient honest prover we know of is the one for the sumcheck protocol, which can compute its responses with an oracle for #P.

Why do we care about the power of the honest prover? We give two applications below.

4.1 Relations Among Complexity Classes

Recall the following theorems.

Theorem 9. *If* $\mathrm{NP} \subseteq \mathrm{P/poly}$ *then* $\mathrm{PH} = \Sigma_2^p$.

Theorem 10. *If* $\mathrm{PSPACE} \subseteq \mathrm{P/poly}$ *then* $\mathrm{PSPACE} = \Sigma_2^p$.

Theorem 11. *If* $\mathrm{EXP} \subseteq \mathrm{P/poly}$ *then* $\mathrm{EXP} = \Sigma_2^p$.

We proved Theorem 9 in an earlier lecture and left Theorems 10 and 11 as exercises. We can now prove the following stronger version of Theorem 10.

Theorem 12. *If* $\mathrm{PSPACE} \subseteq \mathrm{P/poly}$ *then* $\mathrm{PSPACE} = \mathrm{MA}$.

Proof. Recall that in the last lecture we proved that MA $\subseteq \Sigma_2^p$, so this is indeed a stronger result. Let L be a language in PSPACE. We know that there is an interactive proof system for L in which the honest prover's responses are computable in polynomial space. Assuming PSPACE $\subseteq$ P/poly, there is a polynomial-size circuit implementing the honest prover's strategy (i.e., given a transcript of a message history, it computes each bit of the honest prover's next message). We have Merlin send this circuit to Arthur, who can then carry out the protocol by himself, evaluating the circuit to determine the prover's responses. If the input is in the language then, as we argued above, there is a circuit that makes Arthur accept with probability 1. If the input is not in the language then no prover, and in particular no polynomial-size circuit, can make Arthur accept with probability greater than the soundness of the interactive proof system for L. This proves that $L \in$ MA. □

Since the power of interactive proof systems only extends as far as PSPACE, the above argument does not immediately yield an analogous strengthening of Theorem 11. However, the power of multiple prover interactive proof systems allows us to prove the desired result in much the same way.

Theorem 13. *If* EXP $\subseteq$ P/poly *then* EXP = MA.

Proof of Theorem 13. Let L be a language in EXP. Theorem 3 implies $L \in$ MIP, and it turns out that the strategy of the honest provers for this protocol can be implemented in exponential time. Assuming EXP $\subseteq$ P/poly, there exist polynomial-size circuits implementing the strategy of each of the honest provers in this multiple prover interactive proof system. Merlin sends these circuits to Arthur, who then carries out the protocol by himself, using the circuits to determine the provers' responses. This establishes that $L \in$ MA. □

Natural questions that arise are whether analogous results hold for NEXP and NP. We cannot use this technique to establish an analogous result for NEXP since it is open whether the honest provers for the NEXP protocol given by Theorem 3 can compute their responses in polynomial time given oracle access to NEXP. The question of whether an analogous result holds for NP is also open.

Conjecture 1. *If* NP $\subseteq$ P/poly *then* PH = MA.

Given a language in coNP, if we knew an honest prover for the interactive proof system given by Theorem 1 or Theorem 2 could compute its responses in polynomial time with an oracle for NP, then an almost identical argument to the proof of Theorem 12, together with the fact that coNP $\subseteq$ MA implies PH = MA, would establish Conjecture 1. In fact, it would suffice for the honest prover to use an oracle for any language in PH, since we know that NP $\subseteq$ P/poly implies PH $\subseteq$ P/poly. However, the best honest provers we know of for languages in coNP require the power of counting.

4.2 Program Verification

As a second application of the power of the honest prover, we study the problem of verifying correctness of programs. The program verification problem in its full generality is undecidable, so we set a more modest goal. We are interested in verifying the correctness of a program P on a specific input x, by running P on x and related inputs. It's not reasonable to expect the verification procedure to detect that P is incorrect when it's only wrong on inputs totally unrelated to x. We also want the verification procedure to use less resources than the trivial procedure that solves the problem from scratch and explicitly checks whether P is correct on x. In our model, we assume oracle access to P, so we are not charged for running it as a black box procedure. Formally, we have the following definition.

Definition 3. *An* instance checker *with completeness c and soundness $s < c$ for a language L is a probabilistic polynomial-time oracle Turing machine C such that for all x, if $P = L$ (i.e., the program is correct) then* $\Pr[C^P(x)$ *accepts*$] \geq c$*, and if $P(x) \neq L(x)$ (i.e., the program is incorrect* on the given input*) then* $\Pr[C^P(x)$ *accepts*$] \leq s$*.*

Typical settings of the completeness and soundness parameters are $c = 1$, $s = 1/2$.

We can now demonstrate a very strong connection between instance checkers and probabilistically checkable proof systems.

Theorem 14. *A language L has an instance checker with completeness c and soundness s if and only if both L and $\overline{L}$ have a probabilistically checkable proof system with completeness c and soundness s such that the honest prover's responses are computable in polynomial time with oracle access to L.*

Proof. We first argue the forward direction. Assume that L has an instance checker with completeness c and soundness s. Note that this implies that $\overline{L}$ has an instance checker with the same parameters; it is the same as the instance checker for L but it flips the answers to the queries it makes. Thus by symmetry, it suffices to show that L has a probabilistically checkable proof system with the same parameters such that an honest prover's responses are computable in polynomial time with oracle access to L. We simply run the instance checker on our input x and use our proof to answer the instance checker's queries. The correct proof consists of the correct answers to the queries to L. When $x \in L$, completeness follows by the completeness of the instance checker. For $x \notin L$, we would like to use the soundness property of the instance checker. This only holds for oracles P with $P(x) \neq L(x)$, so we have the verifier first ask if x is in L before running the instance checker. If the proof claims $x \notin L$, we reject immediately. If the proof claims $x \in L$, we proceed as before. Completeness is preserved, and now for the case when $x \notin L$, a cheating proof is forced to falsely claim that $x \in L$ to get the verifier to accept. Then soundness follows by the soundness of the instance checker.

Now we argue the other direction. Assume that L and $\overline{L}$ have probabilistically checkable proof systems as in the statement of the theorem. Then by Theorem 6, each language has a multiple prover interactive proof system with at least as good of parameters and where the honest provers' responses are computable in polynomial time with oracle access to the language. We design an instance checker for L. On input x, we ask our oracle whether $x \in L$. If the oracle says yes, then we run the multiple prover interactive proof system for L and otherwise we run the one for $\overline{L}$. In simulating one of the verifiers, we respond to queries by running the appropriate honest prover's strategy using our oracle. We accept if and only if the verifier we're simulating accepts. Suppose that our oracle agrees with L. Then we run a multiple prover interactive proof system such that x is in the language decided by it, and all the responses to the prover queries agree with the honest prover's strategy, so we accept with probability at least c. Suppose that our oracle disagrees with L on x. Then we run the wrong multiple prover interactive proof system; x is not in the language decided by it. In this case, it doesn't matter how we respond to the prover queries; we accept with probability at most s. $\square$

Corollary 3. *There exist instance checkers with perfect completeness for Graph Nonisomorphism, every* PSPACE*-complete language, every* #P*-complete function (provided we extend our definition of instance checker to function problems in the natural way), and every* EXP*-complete language.*

Proof. We noted at the beginning of Section 4 that the honest prover's strategy for the private-coin interactive proof system for Graph Nonisomorphism can be implemented in polynomial time with an oracle for Graph Nonisomorphism. This protocol also has perfect completeness. There is trivially an interactive proof system with perfect completeness and soundness 0 for Graph Isomorphism, where the honest prover's strategy is just the problem of finding an isomorphism between two graphs if one exists. As noted in a previous lecture, this problem is solvable in polynomial time with oracle access to (the decision version of) Graph Isomorphism. The first result now follows from the proof of Theorem 14 (bypassing the probabilistically checkable proof systems in the statement of the theorem and directly applying the above interactive proof systems).

Similarly, the result holds for every PSPACE-complete language using the interactive proofs for PSPACE developed in Section 2 and the fact that all polynomial-space computations can be carried out in polynomial time with an oracle for a PSPACE-complete language.

The third result holds using an argument similar to the proof of Theorem 14 where the instance checker first asks its oracle what the value of the function is on the given input x, then parsimoniously reduces x to a #SAT formula and runs the sumcheck protocol. The honest prover's strategy can be implemented in polynomial time with oracle access to #SAT and hence with oracle access to L by completeness.

For the final result, recall from the proof of Theorem 13 that every EXP-complete language has a multiple prover interactive proof system where the honest provers' responses are computable in exponential time and hence in polynomial time with oracle access to an EXP-complete language. This protocol also has perfect completeness. Since EXP is closed under complement, the result then follows from the proof of Theorem 14. □