PRINCETON UNIVERSITY COS 522: COMPUTATIONAL COMPLEXITY

Lecture 6: Interactive Proofs - cont.

Lecturer: *Sanjeev Arora* Scribe: *Elad Hazan*

1 Preliminaries about polynomials

(Notes borrowed from Sanjeev's past lecture notes.)

THEOREM 1
A nonzero polynomial of degree d has at most d distinct roots.

PROOF: Suppose $p(x) = \sum_{i=0}^{d} c_i x^i$ has $d+1$ distinct roots $\alpha_1, \ldots, \alpha_{d+1}$ in some field F. Then

$$\sum_{i=0}^{d} \alpha_j^i \cdot c_i = p(\alpha_j) = 0,$$

for $j = 1, \ldots, d+1$. This means that the system $\mathbf{Ay} = \mathbf{0}$ with

$$\mathbf{A} = \begin{pmatrix} 1 & \alpha_1 & \alpha_1^2 & \ldots & \alpha_1^d \\ 1 & \alpha_2 & \alpha_2^2 & \ldots & \alpha_2^d \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & \alpha_{d+1} & \alpha_{d+1}^2 & \ldots & \alpha_{d+1}^d \end{pmatrix}$$

has a solution $\mathbf{y} = \mathbf{c}$. The matrix $\mathbf{A}$ is a *Vandermonde* matrix, hence

$$\det \mathbf{A} = \prod_{i>j} (\alpha_i - \alpha_j),$$

which is nonzero for distinct α_i. Hence $\operatorname{rank} \mathbf{A} = d+1$. The system $\mathbf{Ay} = \mathbf{0}$ has therefore only a trivial solution — a contradiction to $\mathbf{c} \neq \mathbf{0}$. □

THEOREM 2
For any set of pairs $(a_1, b_1), \ldots, (a_{d+1}, b_{d+1})$ there exists a unique polynomial $g(x)$ of degree at most d such that $g(a_i) = b_i$ for all $i = 1, 2, \ldots, d+1$.

PROOF: The requirements are satisfied by *Lagrange Interpolating Polynomial*:

$$\sum_{i=1}^{d+1} b_i \cdot \frac{\prod_{j \neq i}(x - a_j)}{\prod_{j \neq i}(a_i - a_j)}.$$

If two polynomials $g_1(x), g_2(x)$ satisfy the requirements then their difference $p(x) = g_1(x) - g_2(x)$ is of degree at most d, and is zero for $x = a_1, \ldots, a_{d+1}$. Thus, from the previous theorem, polynomial $p(x)$ must be zero and polynomials $g_1(x), g_2(x)$ identical. □

2 Permanent ∈ IP

DEFINITION 2.1 *Let $A \in F^{n \times n}$ be a matrix over the field F. The permanent of A is:*

$$perm(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i,\sigma(i)}$$

The problem of calculating the determinant is $\#P$-complete (notice the similarity to the determinant which is polynomial time computable). Recall from previous lectures that $PH \subseteq P^{PERM}$ (Toda's theorem).
Observation:

$$f(x_1, x_2, ..., x_n) := perm \begin{bmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,n} \\ x_{2,1} & \ddots & \dots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \dots & x_{n,n} \end{bmatrix}$$

is a degree n polynomial since

$$f(x_1, x_2, \dots, x_n) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} x_{i,\sigma(i)}.$$

We now show two properties of the permanent problem. The first is called ransom self reducibility:

THEOREM 2.2 (LIPTON 88') *There is a randomized algorithm that, given an oracle that can compute the permanent on $1 - \frac{1}{3n}$ fraction of the inputs in $F^{n \times n}$ (where the finite field F has size $> 3n$), can compute the permanent on all inputs correctly with high probability.*

PROOF: Let A be some input matrix. Pick a random matrix $R \in_R F^{n \times n}$ and let $B(x) := A + x \cdot R$ for a variable x. Notice that:

- $f(x) := perm(B)$ is a degree n univariate polynomial.
- For any fixed $b \neq 0$, $B(b)$ is a random matrix, hence the probability that oracle computes $perm(B(b))$ correctly is at least $1 - \frac{1}{3n}$.

Now the algorithm for computing the permanent of A is straightforward: query oracle on all matrices $\{B(i) | 1 \leq i \leq n+1\}$. According to the union bound, with probability of at least $1 - \frac{n+1}{n} \approx \frac{2}{3}$ the oracle will compute the permanent correctly on all matrices.
Recall the fact from numerical analysis that given $n+1$ values $\{(a_i, b_i) | i \in [n+1]\}$, there exists a unique a degree n polynomial p that satisfies $\forall i \ . \ p(a_i) = b_i$.
Therefore, given that the values $B(i)$ are correct, the algorithm can interpolate the polynomial $B(x)$ and compute $B(0) = A$. □

Note: The above theorem can be strengthened to be based on the assumption that the oracle can compute the permanent on a fraction of $\frac{1}{2} + \varepsilon$ for any constant $\varepsilon > 0$ of the inputs. The observation is that not all values of the polynomial must be correct for unique interpolation. There are techniques which require only a fraction slightly higher then half of correct values for successfully interpolating a polynomial. This was not done in class but we include notes from a previous course that describe how; see the last section of this lecture's notes.

Another property of the permanent problem, called downward self reducibility:

$$perm(A) = \sum_{i=1}^{n} perm(A_{1,i})$$

Where $A_{1,i}$ is a $(n-1) \times (n-1)$ sub-matrix of A obtained by removing the 1'st row and i'th column of A (recall the analogous formula for the determinant uses alternating signs).

DEFINITION 2.3 *Define a $(n-1) \times (n-1)$ matrix $D_A(x)$, such that each entry contains a degree n polynomial. This polynomial is uniquely defined by the values of the matrices $\{A_{1,i} | i \in [n]\}$. That is:*

$$\forall i \in [n] \ . \ D_A(i) = A_{1,i}$$

Where $D_A(i)$ is the matrix $D_A(x)$ with i substituted for x. (notice that these equalities force n points and values on them for each polynomial at a certain entry of $D_A(x)$, and hence according to the previously mentioned fact determine this polynomial uniquely)

Observation: $Perm(D_A(x))$ is a degree $n(n-1)$ polynomial in x.
We now show an interactive proof for the permanent (the decision problem is whether $perm(A) = k$ for some value k):

- Round 1: Prover sends to verifier a polynomial $g(x)$ of degree $n(n-1)$, which is supposedly $perm(D_A(x))$.
- Round 2: Verifier checks whether:
$$k = \sum_{i=1}^{m} a_{1,i} g(i)$$
If not, rejects at once. Otherwise, verifier picks a random element of the field $b_1 \in_R F$ and asks the prover to prove that $g(b_1) = perm(D_A(b_1))$. This reduces the matrix dimension to $(n-2) \times (n-2)$.

$$\vdots$$

- Round $2(n-1)-1$: Prover sends to verifier a polynomial of degree 2, which is supposedly the permanent of a 2×2 matrix.
- Round $2(n-1)$: Verifier is left with a 2×2 matrix and calculates the permanent of this matrix and decides appropriately.

Claim 2.4 *This is indeed an interactive protocol.*

Proof: If $perm(A) = k$, then there exists a prover that makes the verifier accept with probability 1, this prover just returns the correct values of the polynomials according to definition.
On the other hand, suppose that $perm(A) \neq k$. If on the first round, the polynomial $g(x)$ sent is the correct polynomial $D_A(x)$, then:
$$k \neq \sum_{i=1}^{m} a_{1,i} g(i) = perm(A)$$
And the verifier would reject. Hence $g(x) \neq D_A(x)$. According to the fact on polynomials stated above, these polynomials can agree on at most $n(n-1)$ points. Hence, the probability that they would agree on the randomly chosen point b_1 is at most $\frac{n(n-1)}{|F|}$. The same considerations apply to all subsequent rounds if exist, and the overall probability that the verifier will not accepts is thus (assuming $|F| \geq 10n^3$ and sufficiently large n):
$$\begin{aligned} Pr &\geq \left(1 - \frac{n(n-1)}{|F|}\right) \cdot \left(1 - \frac{(n-1)(n-2)}{|F|}\right) \cdot ... \left(1 - \frac{3 \cdot 2}{|F|}\right) \\ &\geq \left(1 - \frac{n(n-1)}{|F|}\right)^{n-1} \geq (1 - o(1)) e^{-2\frac{n(n-1)^2}{|F|}} \geq \frac{1}{2} \end{aligned}$$
□

Open Problem: The current technique to obtain an interactive protocol for $\overline{SAT}$ (the complement of the SAT language) is to reduce $\overline{SAT}$ to permanent and then obtain the IP for permanent we've seen. The power of the prover is, hence, $\#P$ (can compute a $\#P$ complete language). An open problem is whether there exists an IP for $\overline{SAT}$ with less power, specifically can the prover be in FP^{SAT} ?

3 Program Checking

The first interactive proof for permanent (outlined an email Noam Nisan sent to many researchers in 1989) was motivated by a field called *program checking,* initiated by Blum and Kannan. A *program checker* C for a program P is itself another program. For any instance on which P is run, C is run subsequently and certifies that P is correct or declares it to be incorrect. Formally:

DEFINITION 3.1 *Let P be a claimed program for computational task T. A* ***checker*** *for T is a probabilistic polynomial time TM, C, such that:*

1. *If $\forall x$. $P(x) = T(x)$, then $P[C^P$ accepts $P(x)] \geq \frac{2}{3}$*
2. *If $P(x) \neq T(x)$ then $P[C^P$ accepts $P(x)] < \frac{1}{3}$*

EXAMPLE 1 (GRAPH NON-ISOMORPHISM) The input for the problem of Graph Non-Isomorphism is a pair of labelled graphs $< G_1, G_2 >$, and the problem is to decide whether a permutation $\pi : V(G_2) \mapsto V(G_2)$ such that renaming the vertices according to the permutation results in a graph identical to G_1 (and in that case the graphs are isomorphic, denoted $G_1 \equiv G_2$).

Now there are two cases, depending upon whether or not the program claims $G_1 \equiv G_2$. If it claims that $G_1 \equiv G_2$ then the checking the claim is left as an exercise. We show how to check the claim that $G_1 \not\equiv G_2$ using our earlier interactive proof of Graph non-isomorphism.

Recall the IP for Graph Non-Isomorphism:

- In case prover admits $G_1 \not\equiv G_2$ repeat k times:
- Choose $i \in_R \{1, 2\}$. Permute G_i randomly into H
- Ask the prover $< G_1, H >; < G_2, H >$ and check to see if the prover's first answer is consistent.

Given a computer program that supposedly computes graph isomorphism, P, how would we check its correctness? The program checking approach suggests to use an IP while regarding the program as the prover. Let C be a program that performs the above protocol with P as the prover, then:

THEOREM 3.2 *If P is a correct program for Graph Non-Isomorphism then C outputs "correct" always. Otherwise, if $P(G_1, G_2)$ is incorrect then $P[C$ outputs "correct" $] \leq 2^{-k}$. Moreover, C runs in polynomial time.*

Program checking is sometimes also called *instance checking*, perhaps a more accurate name since checking a program would mean that it computes T on *all inputs*, which is of course undecidable. Blum and Kannan's definition manages to evade this undecidability result by concentrating on the correctness of the program on a single instance.

3.1 Program Checking Properties

The following theorem are a direct consequence of the IP we have seen:

THEOREM 3.3 *The problems Graph Non-Isomorphism, Quadratic Residue, Permanent have checkers.*

The following theorem shows that program checking is a robust notion:

THEOREM 3.4 (BLUM,KANNAN,BEIGEL) *Let $\mathcal{F}$ be a time complexity class which is closed under multiplication, composition and addition of its time complexity functions (such as P and $TIME(\log^{O(1)} n)$). Let π_1 and π_2 two problems reducible to each other in $\mathcal{F}$, and C_1 a checker for π_1 in $\mathcal{F}$. Then there is a checker C_2 for π_2 in $\mathcal{F}$.*

PROOF:(sketch) Basically one can produce the new checker using the checker for the first problem composed with a program that computes the reduction function. Correctness follows directly from the definition of reductions and the fact that the complexity class is closed under basic operations. □

Using this theorem, and the fact that P complete languages are NC-reducible to each other, it suffices to show a checker in NC for one P-complete language (as was shown by Blum & Kannan) to obtain the following interesting fact:

THEOREM 3.5 *For any P-complete language there exists a program checker in NC.*

The relation between program checking and IP is more precisely stated by the following theorem:

DEFINITION 3.6 *Let **function restricted IP** the set of all decision problems for which there exists an IP, where the prover is further restricted to be a function from the program instances to* $\{YES/NO\}$.

THEOREM 3.7 *A language L has a checker $\Leftrightarrow$ $L \in$ function restricted IP $\cap$ co-function restricted IP.*

PROOF: $\Leftarrow$ Suppose that L has an IP of one of the sorts above. A checker for the language would simulate the IP, using the supposed machine for the language (or co-language) as provers.
$\Rightarrow$ Suppose the language L has a checker, denoted M. Then a (co-) function restricted IP for this language is just the simulation of the machine, such that each activation of M of the oracle is a question for the prover. Correctness follows simply from definitions. □

Dispersal of Information Using Polynomials

These are Sanjeev's old lecture notes describing how to reconstruct a polynomial from noisy values.

Polynomials are often useful in situations where information needs to be dispersed in an error-tolerant way, so that it can be reconstructed even if it is partially corrupted or destroyed.

Suppose we want to encode a message into a finite number of packets to be transmitted through a faulty network. This network can drop up to 1/2 of packets, but it does not corrupt the contents of the remaining packets. To achieve a successful transmission, we can use polynomial interpolation:

Encoding. Without loss of generality assume the message is a d+1-tuple $c_0, c_1, \ldots, c_d \in F$, where $|F| > 2d$. Take $2d+1$ distinct points $\alpha_1, \ldots, \alpha_{2d+1} \in F$ and determine values of the polynomial $p(x) = \sum_{i=0}^{d} c_i x^i$ at α_i. Send packets $(\alpha_1, p(\alpha_1)),\ldots,(\alpha_{2d+1}, p(\alpha_{2d+1}))$.

Decoding. Packets describe the polynomial p with sufficient redundancy. Even when d packets are dropped, the polynomial p and hence the original message is uniquely determined by the remaining $d+1$ pairs. This observation was used in Lipton's theorem (Theorem 2.2). Now suppose that the network can corrupt up to 1/4th of the packets. We will use a strategy developed by Berlekamp and Welch in 1985. In order to transmit a message described by a polynomial $p(x)$ of degree d, we will send $20d$ pairs $(\alpha_i, p(\alpha_i))$. Let the received pairs be (α_i', β_i') (for missing packets, we can set $\alpha_i' = \beta_i' = 0$). A pair (α_i', β_i') will be considered corrupted if $p(\alpha_i') \neq \beta_i'$. Then there exists a *nonzero* polynomial $e(x)$ of degree at most $5d$, which is zero at all corrupted values α_i' — this is called an *error locator polynomial*.

LEMMA 3
There exist nonzero polynomials $e(x)$ and $c(x)$ such that

$$\deg e \leq 5d$$
$$\deg c \leq 6d$$

and $c(\alpha_i') = \beta_i' e(\alpha_i')$ for $i = 1, 2, \ldots, 20d$.

PROOF: Taking the error locator polynomial $e(x)$ and $c(x) = p(x)e(x)$ we obtain

$$c(\alpha_i') = p(\alpha_i')e(\alpha_i') = \begin{cases} \beta_i' e(\alpha_i') & \text{if pair } (\alpha_i', \beta_i') \text{ is not corrupted} \\ p(\alpha_i') \cdot 0 = 0 = \beta_i' \cdot 0 & \text{if pair } (\alpha_i', \beta_i') \text{ is corrupted.} \end{cases}$$

□

COROLLARY 4
Polynomials $e(x)$ and $c(x)$ that satisfy conditions of previous lemma can be found in time polynomial in d.

PROOF: Equations $c(\alpha'_i) = \beta'_i e(\alpha'_i)$, where coefficients of c and e are unknown, form a system of $20d$ linear equations in $11d + 2$ unknowns. The lemma guarantees its feasibility. We can solve for coefficients by Gaussian elimination. □

THEOREM 5 (BERLEKAMP-WELCH)
If $c(x), e(x)$ satisfy the conditions of the lemma then $e(x)|c(x)$ and $p(x) = c(x)/e(x)$.

PROOF: Consider the polynomial $c(x) - p(x)e(x)$. It has degree at most $6d$, but it has at least $15d$ roots because it is zero on all noncorrupted α'_i's. Therefore, $c(x) - p(x)e(x) \equiv 0$ and $c(x) \equiv p(x)e(x)$. □

REMARK 1 This strategy will work whenever a fixed δ-fraction of packets is corrupted, where $\delta < 1/2$. Somebody asked if a scheme is known that recovers the polynomial even if more than $1/2$ the packets are corrupted. The answer is Yes, using Sudan's list decoding algorithm.