6.842 Randomness and Computation February 11, 2020

Lecture 3

Lecturer: Amartya Shankha Biswas *Scribe: Lillian Zhang*

1 Introduction

Today we'll first present a randomized algorithm for Polynomial Identity Testing with two applications: (1) checking the equality of two strings and (2) checking if a perfect bipartite matching exists. Next, we will present an algorithm to uniformly generate satisfying assignments to a DNF formula, which will be used next class in approximately counting of the number of satisfying assignments to a DNF formula.

2 Polynomial Identity Testing (PIT)

Definition 1 Polynomial Identity Testing (PIT) *is the following problem: Given two polynomials p and q in n variables, we want to determine if they are the same, i.e. is $p(x_1, x_2, \ldots, x_n) = q(x_1, x_2, \ldots, x_n)$ for all $x_1 \ldots x_n$.*

For example, consider

$$(x_1+x_2)(x_3+x_4)^{40}(x_5^2+x_6)\overset{?}{\equiv}(x_1-x_2)(x_3-x_4)^{40}(x_5^2+x_6)+(x_1+2x_2)(x_3-x_4)^{40}(x_5^2+x_6)+x_1x_3x_5^2+x_2x_6x_3^{40}.$$

We would like to figure out if the polynomial on the left is identically equal to the polynomial on the right. The problem with opening up the brackets and expanding is that we could possibly get an exponential (in the degree of the polynomial) number of terms. For instance, if there are n variables and the degree of d, there could be more that $\binom{n}{d}$ terms!

Consider the following related problem of *Polynomial Zero Testing*, which we will show is equivalent to Polynomial Identity Testing.

Definition 2 Polynomial Zero Testing (PZT) *is the following problem: Given a polynomial $p(x_1, \ldots, x_n)$, we want to determine if it's identically 0, i.e. is $p(x_1, x_2, \ldots, x_n) = 0$ for all $x_1 \ldots x_n$.*

Polynomial Zero Testing is a special case of Polynomial Identity Testing when q is the zero polynomial. Further, any PIT problem can be re-framed as a PZT problem. To determine if $p(x_1, x_2, \ldots, x_n) = q(x_1, x_2, \ldots, x_n)$, we may instead consider if $(p-q)(x_1, x_2, \ldots, x_n)$ is identically 0, as $p - q \equiv 0$ iff $p \equiv q$.

Thus an algorithm for PZT *is* an algorithm for PIT, but before presenting the algorithm, let us clarify some notions.

Assume that the *domain is a field* such as $\mathbb{R}$ or $\mathbb{Z}_p$. For example, if the field is $\mathbb{Z}_7$, then

$$(x+3)^2 \equiv x^2 + 6x + 9 \equiv x^2 + 6x + 2 \pmod 7.$$

The *degree of a univariate polynomial* is the highest exponent of a term in the polynomial. For instance, the degree of $x^{10} + x^3 + 1$ is 10. The *total degree of a multivariate polynomial* is the max over all terms of the sum of degrees in the term. This is the notion we'll tend to use most of the time unless specified otherwise. The *maximum degree of a multivariate polynomial* is the maximum over all terms of the degree of the maximum degree variable in that term. For example, if the polynomial is $xy^2 + x^2$, then the total degree is 3, and the maximum degree is 2.

2.1 Algorithm for Polynomial Zero Testing

Assume a polynomial p in n variables and of degree d is given as a black-box oracle. The oracle takes as input $\bar{x} = (x_1, \ldots, x_n)$ and outputs $p(\bar{x})$.

2.1.1 Deterministic Algorithm for the Case when p is Univariate.

Consider a strategy where we plug a value x into the black-box. If $p(x)$ is nonzero, then p is nonzero. If $p(x)$ is zero, then p is zero or x is a root of p. Since a non-zero polynomial of degree d can have at most d roots, then the following is a deterministic algorithm for this problem using $O(d)$ evaluations:

Plug in any $d+1$ distinct values to the black-box. If all are 0, then output "$\equiv$0". Else, output "$\not\equiv$0".

2.1.2 Randomized Algorithm for the Univariate Case.

We can randomize the above algorithm and use only $O(1)$ evaluations as follows:

Pick an element uniformly at random from a set $S \subseteq F$ of size at least $2d$ and plug it into the black-box. If it outputs 0, then output "$\equiv$0". Else, output "$\not\equiv$0".

Since at most $d/|S| \leq 1/2$ of the fraction of all field elements are zeroes of the polynomial, then if $p \not\equiv 0$, Pr[algorithm outputs "$\equiv$0"] $\leq d/|S| \leq 1/2$. Thus if $p \equiv 0$, the algorithm outputs "$\equiv 0''$. If $p \not\equiv 0$, Pr[algorithm outputs "$\not\equiv$0"] $\geq 1/2$.

Technical note: There is a very unlikely but unfortunate case where we pick a field such that p is identically 0 in the field, but not in $\mathbb{Z}$, such as if the coefficients of p are multiples of a prime n and we restrict to $\mathbb{Z}/n\mathbb{Z}$, the field of integers modulo n. Using multiple fields would alleviate this problem.

This same idea is generalized to give an algorithm that works in the multivariate case.

2.1.3 Randomized Algorithm for the Multivariate Case. (Schwartz Zippel Lemma)

Observe that a multivariate polynomial can have infinitely many roots - for example: $p(x, y) = x \cdot y$ over a field of infinite size has infinitely many roots. However, the same kind of test that works in the univariate case works in this case so as long as the field is of large enough size, and a random value is picked for each x_i.
The following is the algorithm that works in the multivariate case (for total degree d, needs $|F| \geq 2d$):

1. Pick $S \subseteq F$ arbitrarily such that $|S| \geq 2d$.
2. Pick uniformly at random $x_1, x_2, \ldots, x_n \in_R S$.
3. If $p(x_1, \ldots, x_n) = 0$, output "$\equiv$0"; else output "$\not\equiv$0".

Again, the algorithm needs just $O(1)$ evaluations of p on numbers of size $O(\log d)$.This is shown by the below claim, which can be proved by induction on n, with the univariate case as the base case.

Claim 3 *If the polynomial $p \not\equiv 0$, then* $\Pr[p(x_1, \ldots, x_n) = 0] \leq \frac{d}{|S|}$.

The behavior of the algorithm is as follows. If $p \equiv 0$, the algorithm outputs "$\equiv$0". If $p \not\equiv 0$, Pr[algorithm outputs "$\not\equiv$0"] $\geq 1/2$. This probability can be improved by either repeating the algorithm multiple times, or picking a bigger field size.

2.2 Applications of PIT

Now, we present some applications of both univariate, and multivariate *polynomial identity testing.*

2.2.1 Problem Set Transmission

Assume you at MIT have a problem set represented as a string $\bar{a} = a_1 a_2 \ldots a_n$ and Ronitt at a very fancy seminar very far away has a version of your problem set represented as a string $\bar{b} = b_1 b_2 \ldots b_n$. You want to decide if Ronitt has the right version of your problem set, that is, if $\bar{a} = \bar{b}$.

One way of doing this would be for you to transmit $\bar{a}$ to Ronitt. This would take n bits of transmission.

Alternatively, you could view $\bar{a}$ as the coefficients of a degree n univariate polynomial $p = \sum a_i x^i$ over a field of size at least $2n$. Then, pick arbitrarily $S \subseteq F$ such that $|S| \geq 2n$, pick $x \in_R S$, and send x and $\bar{a}(x)$ to Ronitt. She checks if $\bar{a}(x) = \bar{b}(x)$. If yes, you conclude that $\bar{a} = \bar{b}$. Else they're different.

By our previous analysis, the above algorithm indeed works. If $\bar{a} = \bar{b}$ then you will always conclude that they are equal, and if $\bar{a} \neq \bar{b}$, then with probability at least a half, you would find that out. In the above randomized algorithm, only $O(\log n)$ bits are needed to be transmitted.

2.2.2 Bipartite Matching

We'll first introduce some preliminaries. A *bipartite graph* $G = (V, E)$ is one in which the set of vertices V can be partitioned into two sets S and T such that all the edges in G go between S and T, and there are no edges with both endpoints lying in the same set. A *matching* M is a subset of the set of edges E such that no two edges in M share a vertex. A *perfect matching* is one that has an edge adjacent to each vertex.

A natural question that arises in graph theory is that given a graph, decide if it has a perfect matching. The problem of actually finding a perfect matching in a graph that has one can be solved using network flows. The result we'll show here is not as strong, but is a first step in the direction of getting the best algorithm for finding a bipartite matching. We'll show how to use Polynomial Zero Testing to give a randomized algorithm for determining if a given graph has a perfect matching.

Given a graph $G = (V, E)$, the *Tutte matrix* $A_G = [a_{ij}]$ of the graph is a matrix whose entries are variables or zeroes. In particular, $a_{ij} = x_{ij}$ if $(i, j) \in E$, and $a_{ij} = 0$ otherwise.

Claim 4 *A graph G has a perfect matching iff* $\det(A_G) \neq 0$.

Proof We begin by writing out the expression for the determinant as a sum of products of entries.

$$\det(A_G) = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{i\sigma_i}$$

Here S_n denotes the set of all permutations of 1 through n. Observe that every permutation represents a possible matching between the two vertex sets. Also, $\prod_{i=1}^{n} a_{i\sigma_i} \not\equiv 0$ if and only if all the edges corresponding the the perfect matching represented by the permutation σ are present in G. Since every other term has a different combination of variables, the non-zero terms don't get canceled out. This completes the proof of the above claim. ■

Determining if there exists a perfect matching Since the determinant of A_G is a multivariable polynomial of degree n, it can be tested by the Polynomial Zero Testing algorithm to see if it's identically zero or not, and hence to determine if the graph G contains a perfect matching. We use Polynomial Zero Testing because the determinant polynomial may give $n!$ terms, which is too many for us to compute quickly. However, after randomly assigning the variables x_{ij} to values, the determinate can be computed quickly, in $O(n^\omega)$ time sequentially, for $\omega \approx 2.38$, and $O(\log^2 n)$ time in parallel with poly(n) processors.

Finding a perfect matching If a bipartite graph has a perfect matching, we can find one such matching by repeatedly removing an edge from the graph and checking if there is still a perfect matching in the graph.

(a) If there are still perfect matchings after the removal of the edge, then we can continue removing and checking edges.

(b) If there are no longer any perfect matchings, then before the removal every perfect matching uses that edge, so we add it to the perfect matching we are building and remove its two endpoints from the graph. Then the new graph again has perfect matchings, so we can continue removing and checking edges.

By the end of this process, we will have built a perfect matching because at every step we are assured that there is a perfect matching, and each removed edge brings us closer to such a matching because when graph we are looking for perfect matchings in has no edges, we have built a perfect matching.

Other applications for this idea The above idea was used by Lovász to determine if a perfect matching exists in a graph. Since then, it has been shown that similar approaches can be extended to the non-bipartite case as well. For instance, Mucha and Sankowski (FOCS 2004) showed that one can find a maximum matching in general graphs in time $O(n^{\omega})$, which is the time required for matrix multiplication. Nick Harvey at MIT (FOCS 2006) showed a simpler algorithm with the same running time.

3 DNF Sampling

Definition 5 *A* DNF *formula $F = F_1 \vee F_2 \vee \cdots \vee F_m$ on m clauses and n variables is a boolean formula that is an "OR of ANDs". Precisely, given n variables $x_1, \ldots, x_n$, each clause F_i is of the form $F_i = y_{j_1} \wedge y_{j_2} \wedge \ldots$, where the y_j are literals x_k or $\overline{x_k}$.*

3.1 Satisfying DNF Formulas

DNF formulas are easy to satisfy, because satisfying the formula reduces to satisfying a single clause F_i. So we can pick a clause that for all i does not have both x_i and $\bar{x}_i$ and to satisfy it we assign x_i to T if x_i is in the clause, and to F otherwise. For instance, to satisfy the formula

$$F = x_1 x_2 \overline{x_3} \vee \overline{x_1} x_2 x_4,$$

we could satisfy the first clause by choosing $x_1 = x_2 = T$, $x_3 = F$. As an aside, note that if the $\vee$ are replaced by XORs $\oplus$, then F becomes a polynomial in the variables over $\mathbb{Z}_2$, and finding satisfying assignments reduces to random polynomial zero-finding.

Our goal is to uniformly randomly generate satisfying assignments of DNF formulas. Not surprisingly, this is closely related to counting the number of such assignments. However, exact answers to this problem are difficult to obtain: the negation of a DNF formula is a CNF formula, *e.g.* $(x \vee y \vee \overline{z}) \wedge (x \vee \overline{x} \vee y)$. CNF formulas are the subject of the famous $3CNF-SAT$ problem, which shows that finding satisfying assignments for CNF formulas with three variables per clause is NP-complete. Since counting the number of satisfying assignments of a DNF formula would reveal the existence of a satisfying assignment of its negation, counting the number of assignments is a problem of class #P.

3.2 A First Idea

To randomly sample from the satisfying assignments of DNF formulas, we can randomize the procedure to satisfy a formula as below:

A Random Sampler

```
Step i: Pick $i \in_R [m]$.
Step ii: Pick a random satisfying assignment $\pi$ of $F_i$ by picking $x_i = T$ for $x_i$ in $F_i$,
         $x_i = F$ for $\bar{x}_i$ in $F_i$, and arbitrary values for each other $x_i$.
```

However, the first step causes the sampler to be biased towards satisfying assignments from clauses with fewer satisfying assignments, and the second step causes the sampler to be biased toward assignments satisfying several different clauses.

Because we want a uniform distribution of outputs, we make two modifications to address these biases.

3.3 Two Refining Ideas

Let S_i be the set of assignments satisfying F_i. Since $|S_i| = 2^{\#\text{ vars not in } C_i}$, we can and should pick $i \in [m]$ with probability proportional to $|S_i|$ to offset the differing sizes of S_i. Further, for any assignment π, we can count c_π, the number of clauses it satisfies, and thus can be returned from, by checking it against every clause. Then when π is selected, we can and should return it with probability $1/c_\pi$ to offset the number of ways it can be reached. If it is not returned, we can repeat the procedure from the beginning.

3.4 An Algorithm for Uniform Generation of Satisfying DNF Assignments

To uniformly randomly generate π satisfying F:

```
Step i: Pick $i \in [m]$ with probability $\frac{|S_i|}{\sum |S_i|}$.
        Then uniformly pick a random satisfying assignment $\pi$ of $F_i$.
Step ii: Compute $c_\pi = |\{j \in \{1, 2, \ldots, m\} : \pi \in S_j\}|$.
         Then toss a coin with bias $1/c_\pi$.
             If the coin is "Heads", OUTPUT $\pi$ and halt.
             Otherwise, restart at step i.
```

3.4.1 Algorithm Correctness

Claim 6 *Algorithm A outputs satisfying assignments uniformly at random.*

Proof of Claim 6: It suffices to show that each loop iteration is equally likely to output all satisfying assignments π. For a given π, as before let $c_\pi = |\{j \in \{1, 2, \ldots, m\} : \pi \in S_j\}|$. By conditional probability,

$$\begin{aligned}
\Pr[\pi \text{ picked in step 1}] &= \sum_{j \in [m] \ s.t. \ \pi \in S_j} \Pr[\text{Step i picks clause } j] \frac{1}{|S_j|} \\
&= \sum_{j \in [m] \ s.t. \ \pi \in S_j} \frac{|S_j|}{\sum |S_j|} \cdot \frac{1}{S_j} \\
&= \sum_{j \in [m] \ s.t. \ \pi \in S_j} (\sum |S_j|)^{-1} \\
&= \frac{c_\pi}{\sum |S_j|}.
\end{aligned}$$

So the probability that this loop iteration actually outputs π is $\frac{1}{c_\pi} \frac{c_\pi}{\sum |S_j|} = \frac{1}{\sum |S_j|}$, which is independent of π. ■

3.4.2 Algorithm Runtime

Claim 7 *The number of loops needed to choose π satisfies*

$$E[\#\ \textit{loops until OUTPUT}] \le m.$$

Proof of Claim 7: For each π examined, we have $c_\pi \le m$, giving $1/c_\pi \ge 1/m$. A coin with bias p has $1/p$ expected runs until it outputs "Heads", so

$$E[\#\text{ loops}] = 1/\textit{bias} \le m.$$

■