CS 466/666 Algorithm Design and Analysis . Spring 2022

Lecture 7: Polynomial identity testing

We have seen that simple algebraic ideas used in hashing have various applications.

Today, we will see other simple algebraic ideas have surprising applications in designing fast and parallel algorithms, and next time we will see another interesting application in the design of distributed network coding.

String Equality [MR 7.4] This is a striking use of randomness in computation.

Imagine a company maintain multiple copies of the same huge data set.

From time to time they would like to check whether the copies are still the same.

We can think of a data set as a binary string of length n, for a very large n.

Problem: Alice has a binary string $a_1 a_2 \dots a_n$. Bob has a binary string $b_1 b_2 \dots b_n$.

They would like to check whether their strings are equal, with as few communications as possible.

Deterministic algorithms: An obvious algorithm is for Alice to send all her bits to Bob, then Bob can check and tells Alice the answer. But it requires $n+1$ bits of communications.

As you might imagine, there are no better ways to do it, and it can be proved that there are no deterministic algorithms that can do better, using techniques developed in communication complexity.

Randomized algorithms: Surprisingly, there is a randomized algorithm for this problem using only $O(\log n)$ bits.

There are different ways to solve the problem.

One way is to consider the polynomials $A(x) = \sum_{i=1}^{n} a_i x^i$ and $B(x) = \sum_{i=1}^{n} b_i x^i$.

Algorithm:
- Alice and Bob agree on a large enough field F (e.g. modular arithmetic over a large enough prime p).
- Alice picks a random element $r \in F$, evaluate $A(r)$ and send r and $A(r)$ to Bob.
- Bob computes $B(r)$, and return "consistent" if $A(r) = B(r)$, and "inconsistent" if $A(r) \neq B(r)$.

That's the algorithm. We need to analyze its success probability.

Analysis: If the two strings are the same, then $A(x) \equiv B(x)$ and the algorithm will always says "consistent".

So, when the algorithm says "inconsistent", it will never make a mistake.

But the algorithm may make mistake when the two strings are not the same and it says "consistent".

We would like to upper bound this error probability.

If the two strings are not the same, then $A(x) \not\equiv B(x)$ but we have chosen r such that $A(r) = B(r)$.

What is this error probability? It only happens when r is a root of $(A-B)(x) = 0$.

Since $A(x)$ and $B(x)$ are polynomials of degree at most n, so is $(A-B)(x)=0$.

There are at most n roots of a degree n polynomial.

Since we pick a random r in $\mathbb{F}$, the error probability is at most $n/|\mathbb{F}|$.

If we choose a large enough field $\mathbb{F}$ so that $|\mathbb{F}| \approx 1000n$, then this probability is at most 0.001.

And the algorithm only needs to send at most $O(\log|\mathbb{F}|) = O(\log n)$ bits.

How should we decrease the error probability?

This is an algorithm with one-sided error, and the standard way is to repeat the algorithm a few times.

In this case, however, a better way is simply to pick a larger field size, e.g. if $|\mathbb{F}| = n^2$,

then the error probability is at most $\frac{1}{n}$, while we still only send $O(\log n)$ bits.

Polynomial identity testing

Now, we consider a generalization of the argument to multivariate polynomials, and it will be our main tool.

We are given a multivariate polynomial $P(x_1, \dots, x_n)$ and the objective is to determine if

$P(x_1, x_2, \dots, x_n) \equiv 0$, i.e. whether the polynomial is equal to the zero polynomial.

Of course, if the polynomial is given explicitly (e.g. $P(x_1,x_2,x_3) = x_1 x_2^2 x_3^3 + 4x_1^6 x_3 + x_2 x_3^2$), then this problem

is straightforward, but the polynomial can also be given compactly (e.g. $P(x_1,x_2,x_3) = (x_1 - x_2^2)(x_3^3 + x_2^4)\dots(x_1^2 + x_3)$,

or $P(x_1,x_2,x_3) = \det\left(\begin{matrix} x_1 & x_3 & x_2+x_4^2 \\ 0 & x_2 - x_5 & 1 \\ x_2^3 & x_1 & x_4^2 x_2 x_3 \end{matrix}\right))$ and the problem becomes difficult.

In general, there is no known deterministic polynomial time algorithm for this problem.

On the other hand, there is a simple randomized algorithm for this problem, by generalizing the idea

that there are not many roots in a low degree polynomial.

In the following, when we talk about finite fields, we can just think of modular arithmetic over prime.

<u>Lemma</u> (Schwartz-Zippel Lemma) Let $Q(x_1,\dots,x_n) \in \mathbb{F}[x_1,\dots,x_n]$ be a multivariate polynomial of total degree d.

Fix any finite set $S \subseteq \mathbb{F}$, and let $r_1,\dots,r_n$ be chosen independenty and uniformly randomly from S.

Then, $\Pr[Q(r_1,\dots,r_n) = 0] \le \frac{d}{|S|}$ if Q is not identically equal to zero.

(The degree of any term is the sum of the exponents, and the total degree is the maximum degree.)

<u>proof</u> We prove by induction. The base case (single variate polynomial) $n=1$ is well known.

To apply induction, we group the terms of $Q(x_1,\dots,x_n)$ based on the degree of x_1 to obtain

$$Q(x_1,\dots,x_n) = \sum_{i=0}^{k} x_1^i \, Q_i(x_2,\dots,x_n),$$

where k is the largest exponent of x_1 and each $Q_i(x_2,\dots,x_n) \not\equiv 0$.

We condition on the event that $x_2=r_2, x_3=r_3, \dots, x_n=r_n$.

Since $Q_k(x_2,\dots,x_n) \not\equiv 0$ and it has $n-1$ variables, by induction, $\Pr(Q_k(x_2,\dots,x_n)=0) \le \frac{d-k}{|S|}$ as the total degree of Q_k is at most $d-k$ (since x_1 is of degree k in the term $x_1^k Q_k(x_2,\dots,x_n)$).

Now, assuming $Q_k(r_2,\dots,r_n) \neq 0$, then $q(x_1) = Q(x_1, r_2, \dots, r_n)$ is a non-zero degree k single variate polynomial.

So, by the base case, $\Pr(q(r_1)=0) \le \frac{k}{|S|}$.

Combining, we have $\Pr(Q(r_1,\dots,r_n)=0)$

$$= \Pr(Q(r_1,\dots,r_n)=0 \mid Q_k(r_2,\dots,r_n)=0) \cdot \Pr(Q_k(r_2,\dots,r_n)=0) + \Pr(Q(r_1,\dots,r_n)=0 \mid Q_k(r_2,\dots,r_n)\neq 0) \cdot \Pr(Q_k(r_2,\dots,r_n)\neq 0)$$

$$\le 1 \cdot \frac{d-k}{|S|} + \frac{k}{|S|} \cdot 1 = \frac{d}{|S|}. \ \square$$

Determinant testing

To check whether the determinant of an $n\times n$ matrix is identically equal to zero, we can pick a large enough prime field of size $\Theta(\text{poly}(n))$. This will ensure that with high probability that the determinant is still nonzero in this field (if originally it is nonzero). Then, we can just substitute random field elements in the variables and compute the numerical value of the determinant in polynomial time. By Schwartz-Zippel, if the determinant is nonzero, then the (success) probability that the numerical value is nonzero is very high.

Bipartite matching

Given a bipartite graph $G=(U,V;E)$ with $U=\{u_1,\dots,u_n\}$ and $V=\{v_1,\dots,v_n\}$, the bipartite matching problem is to find a maximum cardinality subset of vertex-disjoint edges.

A perfect matching is a matching of n edges.

Theorem (Edmonds) Let A be an $n\times n$ matrix with $A_{ij} = \begin{cases} x_{ij} & \text{if } u_iv_j \in E \\ 0 & \text{otherwise} \end{cases}$ where x_{ij} is a variable.

Then G has a perfect matching if and only if $\det(A) \not\equiv 0$, i.e. $\det(A)$ is a non-zero polynomial.

Proof Recall that $\det(A) = \sum_{\text{permutation } \sigma} \text{sign}(\sigma) \cdot A_{1,\sigma(1)} A_{2,\sigma(2)} \cdots A_{n,\sigma(n)} = \sum_{\text{permutation } \sigma} \text{sign}(\sigma) \prod_{i=1}^{n} A_{i,\sigma(i)}$.

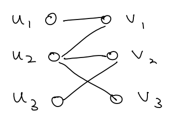
Notice that a permutation defines a bijection between U and V, and so each permutation corresponds to a potential perfect matching between U and V, i.e. $\{u_1v_{\sigma(1)}, u_2v_{\sigma(2)}, \dots, u_nv_{\sigma(n)}\}$.

A permutation σ contributes a non-zero term to $\det(A)$ iff $\{u_1v_{\sigma(1)}, \dots, u_nv_{\sigma(n)}\}$ is a perfect matching in the graph G.

So, if there is no perfect matching in G, then every term is zero and thus $\det(A) \equiv 0$.

On the other hand, if there is a perfect matching in G, then some term is nonzero. Since each term uses a different subset of variables, these terms will not cancel out, and so $\det(A) \not\equiv 0$. $\square$

Pictorically,



$$\begin{array}{c|ccc} & v_1 & v_2 & v_3 \\ \hline u_1 & x_{11} & 0 & 0 \\ u_2 & x_{21} & x_{22} & x_{23} \\ u_3 & 0 & x_{32} & 0 \end{array} \qquad \begin{array}{c|ccc} & v_1 & v_2 & v_3 \\ \hline u_1 & 17 & 0 & 0 \\ u_2 & 33 & 22 & 18 \\ u_3 & 0 & 61 & 0 \end{array}$$

<u>Randomized algorithm</u>: Using Edmond's theorem and the Schwartz-Zippel lemma, we can obtain a "simple" randomized algorithm to check if a bipartite graph has a perfect matching or not.

First, we pick a large enough field $\mathbb{F}$.

We can simply think of doing modular arithmetic over a large prime p.

Then, for each edge $u_iv_j \in E$, set x_{ij} to be a random element in $\mathbb{F}$ (e.g. $\{0, \dots, p-1\}$) independently.

Compute $\det(A)$ over the field $\mathbb{F}$. Return YES if and only if $\det(A) \neq 0$.

<u>Analysis</u>: If there is no perfect matching in G, then $\det(A) = 0$ always, no matter what we substitute.

On the other hand, if there is a perfect matching in G, then $\det(A) \not\equiv 0$ by Edmond's theorem.

Note that $\det(A)$ is a multivariate polynomial of total degree n.

By Schwartz-Zippel lemma, if we put in random values into the variables of a non-zero polynomial, the probability that $\det(A) = 0$ is at most $\frac{n}{|\mathbb{F}|}$.

So, if we set a field size to be at least $100n$ (a prime p at least $100n$), then this probability is at most $\frac{1}{100}$. In other words, the algorithm succeeds with probability at least 0.99.

<u>Time complexity</u>: The most difficult step is to compute the determinant.

Since we put in actual values, the determinant can be computed in $O(n^3)$ time by Gaussian elimination.

There is also an $O(n^\omega)$-time fast matrix multiplication algorithm with $\omega \approx 2.37$.

Theoretically, this is the fastest known algorithm for bipartite matching when the graph is dense.

(As of 2022, this is not true anymore!)

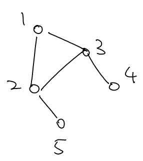
<u>Remark</u>: It is possible to return a perfect matching in the same $O(n^\omega)$ time, although non-trivial.

Also, for the maximum matching problem, we can compute the rank to determine the size.

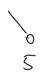
<u>**Non-bipartite matching**</u> (optional)

There is also an algebraic formulation for matching in general graphs.

Given a graph $G = (V, E)$, we consider the following $|V| \times |V|$ matrix A such that for each edge $e = ij$, we have $A_{ij} = x_e$ and $A_{ji} = -x_e$ and other entries to be zero.



$$\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \\ \hline 0 & -x_{12} & x_{13} & 0 & 0 \\ x_{12} & 0 & x_{23} & 0 & -x_{25} \\ -x_{13} & -x_{23} & 0 & x_{34} & 0 \\ 0 & 0 & -x_{34} & 0 & 0 \\ 0 & x_{25} & 0 & 0 & 0 \end{array}$$



$$\begin{bmatrix} 0 & x_{25} & 0 & 0 & 0 \end{bmatrix}$$

Theorem (Tutte) G has a perfect matching if and only if $\det(A) \neq 0$.

Remark: Many combinatorial optimization problems have an algebraic formulation.

Parallel Algorithm and Isolation Lemma

Another advantage of this algebraic approach is to design fast parallel algorithms.

The main reason is that many algebraic problems (such as computing determinant) can be computed efficiently in parallel using a divide-and-conquer approach.

Theorem The determinant of an n by n matrix can be computed in $O(\log^2 n)$ time using $O(n^{\omega+2})$ processors.

Combining this theorem with the algebraic matching algorithm, there is an efficient parallel algorithm to check whether a graph has a perfect matching or not (YES or NO).

We will explain below how to return a perfect matching when there is one.

The difficulty in designing parallel algorithm is to coordinate the processors to search for the same matching.

There are graphs with exponentially many matchings.

The isolation lemma provides a surprising and elegant way to resolve this issue.

Lemma (Isolation Lemma) Given a set system on a ground set of n elements, if we assign a weight independently and uniformly randomly from $\{1, 2, \ldots, 2n\}$, then the probability that there is a unique minimum weight set is at least $1/2$.

Remark: This is quite counter-intuitive. A set system could have $\Omega(2^n)$ sets.

On average, there are $\Omega(2^n/(2n^2))$ sets of a given weight, as the maximum weight is $\leq 2n^2$.

But the isolation lemma says with good probability there is only one set of minimum weight.

Proof Let $\mathcal{F}$ be the set system. Let v be an arbitrary element.

Let $\mathcal{F}_v$ be the set of subsets which contain v. Let $\mathcal{F}_{\bar{v}}$ be the set of subsets which don't contain v.

Consider the quantity $\alpha_v = \min_{S \in \mathcal{F}_{\bar{v}}} w(S) - \min_{R \in \mathcal{F}_v} w(R - v)$.

Note that if $\alpha_v < 0$, then v does not belong to any minimum weight set, as there is a subset S which does not contain v and its weight is smaller than the weight of every subset that contains v (even if the weight of v is not considered yet).

More accurately, by a similar reasoning, if $\alpha_v < w(v)$, then v doesn't belong to any minimum weight set.

On the other hand, if $\alpha_v > w(v)$, then every minimum weight set must contain v.

We say an element v is <u>ambigious</u> if $\alpha_v = w(v)$.

Note that α_v is a quantity independent of $w(v)$, and $w(v)$ is chosen uniformly randomly from $\{1, 2, \ldots, 2n\}$.

Therefore, the probability that v is ambiguous is at most $\frac{1}{2n}$.

By union bound, the probability that some element is ambiguous is at most $\frac{1}{2}$.

Finally, observe that if two different sets A, B have the minimum weight, then any element in $A-B \neq \phi$ and $B-A \neq \phi$ must be ambiguous.

Therefore, we conclude that there are two different sets with the minimum weight is at most $\frac{1}{2}$. □

For the matching problem, we assign each edge an independent random weight from $\{1, 2, \ldots, 2m\}$.

By the isolation lemma, with the set system being all possible perfect matchings, there is a unique minimum weight perfect matching with probability at least $\frac{1}{2}$.

So, now our goal is to output this unique minimum weight perfect matching.

<u>Minimum weight perfect matching</u>

For each edge e, we put the value $2^{w(e)}$ to the variable x_e.

Let the unique minimum weight perfect matching have weight $W \leq 2m^2$.

Then, only one term in the determinant has value 2^W.

All other terms in the determinant have value $2^{W'}$ for $W' > W$, an even number times bigger than 2^W.

<u>Computing the minimum weight</u>

We can search for the minimum weight in parallel, using the following observation: $\frac{\det(A)}{2^k} = \begin{cases} \text{even} & \text{if } k < W \\ \text{odd} & \text{if } k = W \\ \text{not divisible} & \text{if } k > W \end{cases}$.

We can find this k in parallel, after computing $\det(A)$ in parallel.

Note that $\det(A)$ can still be computed efficiently, as each entry has at most $2m$ bits, and this is where the isolation lemma is useful in that it does the job by only using small weights.

<u>Finding the minimum weight perfect matching</u>

For each edge e, in parallel, we want to check whether e belongs to the unique minimum perfect matching.

Let $e = uv$, and G_{uv} be the graph obtained from G by deleting the vertices u and v.

Then, note that $e = uv$ is in the unique minimum weight perfect matching in G if and only if the minimum weight perfect in G_{uv} has weight $W - w(uv)$.

This again can be checked in parallel using different processors for different G_{uv}, and so the minimum weight perfect matching can be constructed efficiently in parallel.

Remark: In general, other algorithms based on algebraic formulations can be made parallel.

Reference: Chapter 7 of "Randomized algorithms" by Motwani and Raghavan.

Open question: Given a graph where each edge is colored red or blue, we would like to determine if there is a perfect matching with exactly k red edges. It's a good exercise to show that there is a randomized polynomial time algorithm for this problem using the algebraic approach, but it has been an open question whether there is a deterministic algorithm for the problem.

Open question: It is a major open question to find a deterministic parallel algorithm for matching. Recently, there is some interesting progress in derandomizing the isolation lemma, which leads to a parallel algorithm using $n^{O(\log n)}$ many processors. Nice question to think about.