# Lecture 07: Algebraic Techniques
# Fingerprinting, Verifying Polynomial Identities, Parallel Algorithms for Matching Problems

Rafael Oliveira

University of Waterloo
Cheriton School of Computer Science

rafael.oliveira.teaching@gmail.com

May 21, 2024

# Overview

- Introduction
  - Why Algebraic Techniques in computer science?
  - Fingerprinting: String equality verification
- Main Problems
  - Polynomial Identity Testing
  - Randomized Matching Algorithms
  - Isolation Lemma
- Remarks
- Acknowledgements

# Why use algebraic techniques in Computer Science?

It is hard to overstate the importance of algebraic techniques in computing.

# Why use algebraic techniques in Computer Science?

It is hard to overstate the importance of algebraic techniques in computing.

- Very useful tool for randomized algorithms (hashing, today's lecture)

# Why use algebraic techniques in Computer Science?

It is hard to overstate the importance of algebraic techniques in computing.

- Very useful tool for randomized algorithms (hashing, today's lecture)
- Parallel & Distributed Computing (this lecture and lecture 23)

# Why use algebraic techniques in Computer Science?

It is hard to overstate the importance of algebraic techniques in computing.

- Very useful tool for randomized algorithms (hashing, today's lecture)
- Parallel & Distributed Computing (this lecture and lecture 23)
- Interactive proof systems (lecture 22)

# Why use algebraic techniques in Computer Science?

It is hard to overstate the importance of algebraic techniques in computing.

- Very useful tool for randomized algorithms (hashing, today's lecture)
- Parallel & Distributed Computing (this lecture and lecture 23)
- Interactive proof systems (lecture 22)
- Efficient proof/program verification (PCP - a bit in lecture 22)
    - Applications in hardness of approximation!
    - Applications in blockchain (Zcash for instance)
    - Zero Knowledge proofs

# Why use algebraic techniques in Computer Science?

It is hard to overstate the importance of algebraic techniques in computing.

- Very useful tool for randomized algorithms (hashing, today's lecture)
- Parallel & Distributed Computing (this lecture and lecture 23)
- Interactive proof systems (lecture 22)
- Efficient proof/program verification (PCP - a bit in lecture 22)
  - Applications in hardness of approximation!
  - Applications in blockchain (Zcash for instance)
  - Zero Knowledge proofs
- Cryptography

# Why use algebraic techniques in Computer Science?

It is hard to overstate the importance of algebraic techniques in computing.

- Very useful tool for randomized algorithms (hashing, today's lecture)
- Parallel & Distributed Computing (this lecture and lecture 23)
- Interactive proof systems (lecture 22)
- Efficient proof/program verification (PCP - a bit in lecture 22)
    - Applications in hardness of approximation!
    - Applications in blockchain (Zcash for instance)
    - Zero Knowledge proofs
- Cryptography
- Coding theory

# Why use algebraic techniques in Computer Science?

It is hard to overstate the importance of algebraic techniques in computing.

- Very useful tool for randomized algorithms (hashing, today's lecture)
- Parallel & Distributed Computing (this lecture and lecture 23)
- Interactive proof systems (lecture 22)
- Efficient proof/program verification (PCP - a bit in lecture 22)
  - Applications in hardness of approximation!
  - Applications in blockchain (Zcash for instance)
  - Zero Knowledge proofs
- Cryptography
- Coding theory
- many more...

# Why use algebraic techniques in Computer Science?

It is hard to overstate the importance of algebraic techniques in computing.

- Very useful tool for randomized algorithms (hashing, today's lecture)
- Parallel & Distributed Computing (this lecture and lecture 23)
- Interactive proof systems (lecture 22)
- Efficient proof/program verification (PCP - a bit in lecture 22)
  - Applications in hardness of approximation!
  - Applications in blockchain (Zcash for instance)
  - Zero Knowledge proofs
- Cryptography
- Coding theory
- many more...

Derandomizing (i.e., obtaining deterministic algorithms) for some of these settings (whenever possible) is *major open problem* in computer science.

# Verifying String Equality

Suppose Alice and Bob each maintain the same large database of information.[1] They would like to check if their databases are *consistent*.

[1] Think of each of them being a server of a company that deals with massive data.

# Verifying String Equality

Suppose Alice and Bob each maintain the same large database of information.[1] They would like to check if their databases are *consistent*.

- Transmission of all data is expensive (*communication complexity setting*)

[1] Think of each of them being a server of a company that deals with massive data.

# Verifying String Equality

Suppose Alice and Bob each maintain the same large database of information.[1] They would like to check if their databases are *consistent*.

- Transmission of all data is expensive (*communication complexity setting*)
- Sending the entire database not feasible

[1] Think of each of them being a server of a company that deals with massive data.

# Verifying String Equality

Suppose Alice and Bob each maintain the same large database of information.[1] They would like to check if their databases are *consistent*.

- Transmission of all data is expensive (*communication complexity setting*)
- Sending the entire database not feasible
- Say Alice's version of database given by bits $(a_1, \ldots, a_n)$ and Bob's version is $(b_1, \ldots, b_n)$

[1]Think of each of them being a server of a company that deals with massive data.

# Verifying String Equality

Suppose Alice and Bob each maintain the same large database of information.[1] They would like to check if their databases are *consistent*.

- Transmission of all data is expensive (*communication complexity setting*)
- Sending the entire database not feasible
- Say Alice's version of database given by bits $(a_1, \ldots, a_n)$ and Bob's version is $(b_1, \ldots, b_n)$
- Deterministic consistency check requires Alice and Bob to communicate $n$ bits (otherwise adversary would know how to change database to make check fail)

[1] Think of each of them being a server of a company that deals with massive data.

# Verifying String Equality

Suppose Alice and Bob each maintain the same large database of information.[1] They would like to check if their databases are *consistent*.

- Transmission of all data is expensive (*communication complexity setting*)
- Sending the entire database not feasible
- Say Alice's version of database given by bits $(a_1, \ldots, a_n)$ and Bob's version is $(b_1, \ldots, b_n)$
- Deterministic consistency check requires Alice and Bob to communicate $n$ bits (otherwise adversary would know how to change database to make check fail)
- Fingerprinting for the rescue!

[1] Think of each of them being a server of a company that deals with massive data.

# Verifying String Equality

Suppose Alice and Bob each maintain the same large database of information.[1] They would like to check if their databases are *consistent*.

- Transmission of all data is expensive (*communication complexity setting*)
- Sending the entire database not feasible
- Say Alice's version of database given by bits $(a_1, \ldots, a_n)$ and Bob's version is $(b_1, \ldots, b_n)$
- Deterministic consistency check requires Alice and Bob to communicate $n$ bits (otherwise adversary would know how to change database to make check fail)
- Fingerprinting for the rescue!

Communication complexity setting, randomized algorithms, need to work with high probability.

[1] Think of each of them being a server of a company that deals with massive data.

# Verifying string equality

Want to check whether strings $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$ equal.

# Verifying string equality

Want to check whether strings $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$ equal.

Fingerprinting mechanism:

# Verifying string equality

Want to check whether strings $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$ equal.

Fingerprinting mechanism:

1. Let $a = \sum_{i=1}^{n} a_i \cdot 2^{i-1}$ and $b = \sum_{i=1}^{n} b_i \cdot 2^{i-1}$

# Verifying string equality

Want to check whether strings $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$ equal.

Fingerprinting mechanism:

1. Let $a = \sum_{i=1}^{n} a_i \cdot 2^{i-1}$ and $b = \sum_{i=1}^{n} b_i \cdot 2^{i-1}$
2. Let $F_p(x) = x \mod p$ be a fingerprinting function, for a prime $p$

# Verifying string equality

Want to check whether strings $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$ equal.

Fingerprinting mechanism:

1. Let $a = \sum_{i=1}^{n} a_i \cdot 2^{i-1}$ and $b = \sum_{i=1}^{n} b_i \cdot 2^{i-1}$
2. Let $F_p(x) = x \mod p$ be a fingerprinting function, for a prime $p$
3. Protocol:

# Verifying string equality

Want to check whether strings $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$ equal.

Fingerprinting mechanism:

1. Let $a = \sum_{i=1}^{n} a_i \cdot 2^{i-1}$ and $b = \sum_{i=1}^{n} b_i \cdot 2^{i-1}$
2. Let $F_p(x) = x \mod p$ be a fingerprinting function, for a prime $p$
3. Protocol:
    1. Alice picks a random prime $p$ and sends $(p, F_p(a))$ to Bob

# Verifying string equality

Want to check whether strings $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$ equal.

Fingerprinting mechanism:

1. Let $a = \sum_{i=1}^{n} a_i \cdot 2^{i-1}$ and $b = \sum_{i=1}^{n} b_i \cdot 2^{i-1}$
2. Let $F_p(x) = x \mod p$ be a fingerprinting function, for a prime $p$
3. Protocol:
   1. Alice picks a random prime $p$ and sends $(p, F_p(a))$ to Bob
   2. Bob checks whether $F_p(a) \equiv F_p(b) \mod p$, sends

$$\begin{cases} 1, & \text{if the values are equal} \\ 0, & \text{otherwise} \end{cases}$$

# Verifying string equality

Want to check whether strings $(a_1, \dots, a_n)$ and $(b_1, \dots, b_n)$ equal.

Fingerprinting mechanism:

1. Let $a = \sum_{i=1}^{n} a_i \cdot 2^{i-1}$ and $b = \sum_{i=1}^{n} b_i \cdot 2^{i-1}$
2. Let $F_p(x) = x \mod p$ be a fingerprinting function, for a prime $p$
3. Protocol:
   1. Alice picks a random prime $p$ and sends $(p, F_p(a))$ to Bob
   2. Bob checks whether $F_p(a) \equiv F_p(b) \mod p$, sends

$$\begin{cases} 1, & \text{if the values are equal} \\ 0, & \text{otherwise} \end{cases}$$

- **Total bits communicated:** $O(\log p)$ bits (dominated by Alice's message)

# Verifying string equality

Want to check whether strings $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$ equal.

Fingerprinting mechanism:

1. Let $a = \sum_{i=1}^{n} a_i \cdot 2^{i-1}$ and $b = \sum_{i=1}^{n} b_i \cdot 2^{i-1}$
2. Let $F_p(x) = x \mod p$ be a fingerprinting function, for a prime $p$
3. Protocol:
   1. Alice picks a random prime $p$ and sends $(p, F_p(a))$ to Bob
   2. Bob checks whether $F_p(a) \equiv F_p(b) \mod p$, sends

$$\begin{cases} 1, & \text{if the values are equal} \\ 0, & \text{otherwise} \end{cases}$$

- **Total bits communicated:** $O(\log p)$ bits (dominated by Alice's message)
- if $(a_1, \ldots, a_n) = (b_1, \ldots, b_n)$ then protocol always right

# Verifying string equality

Want to check whether strings $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$ equal.

Fingerprinting mechanism:

1. Let $a = \sum_{i=1}^{n} a_i \cdot 2^{i-1}$ and $b = \sum_{i=1}^{n} b_i \cdot 2^{i-1}$
2. Let $F_p(x) = x \mod p$ be a fingerprinting function, for a prime $p$
3. Protocol:
   1. Alice picks a random prime $p$ and sends $(p, F_p(a))$ to Bob
   2. Bob checks whether $F_p(a) \equiv F_p(b) \mod p$, sends

$$\begin{cases} 1, & \text{if the values are equal} \\ 0, & \text{otherwise} \end{cases}$$

- **Total bits communicated:** $O(\log p)$ bits (dominated by Alice's message)
- if $(a_1, \ldots, a_n) = (b_1, \ldots, b_n)$ then protocol always right
- what happens when they are different?

# Verifying string equality

- If $(a_1, \ldots, a_n) \neq (b_1, \ldots, b_n)$, then $a \neq b$.

# Verifying string equality

- If $(a_1, \ldots, a_n) \neq (b_1, \ldots, b_n)$, then $a \neq b$.
- For how many primes can $F_p(a) \equiv F_p(b)$? (i.e., protocol will fail)

# Verifying string equality

- If $(a_1, \ldots, a_n) \neq (b_1, \ldots, b_n)$, then $a \neq b$.
- For how many primes can $F_p(a) \equiv F_p(b)$? (i.e., protocol will fail)
- If a number $M$ is in $\{-2^n, \ldots, 2^n\}$, then number of distinct primes $p \mid M$ is $< n$.

# Verifying string equality

- If $(a_1, \ldots, a_n) \neq (b_1, \ldots, b_n)$, then $a \neq b$.
- For how many primes can $F_p(a) \equiv F_p(b)$? (i.e., protocol will fail)
- If a number $M$ is in $\{-2^n, \ldots, 2^n\}$, then number of distinct primes $p \mid M$ is $< n$.
  1. Each prime divisor of $M$ is $\geq 2$, so if $M$ has $t$ distinct prime divisors, then $|M| > 2^t$

# Verifying string equality

- If $(a_1, \ldots, a_n) \neq (b_1, \ldots, b_n)$, then $a \neq b$.
- For how many primes can $F_p(a) \equiv F_p(b)$? (i.e., protocol will fail)
- If a number $M$ is in $\{-2^n, \ldots, 2^n\}$, then number of distinct primes $p \mid M$ is $< n$.
  1. Each prime divisor of $M$ is $\geq 2$, so if $M$ has $t$ distinct prime divisors, then $|M| > 2^t$
  2. $|M| \leq 2^n \Rightarrow t \leq n$

# Verifying string equality

- If $(a_1, \ldots, a_n) \neq (b_1, \ldots, b_n)$, then $a \neq b$.
- For how many primes can $F_p(a) \equiv F_p(b)$? (i.e., protocol will fail)
- If a number $M$ is in $\{-2^n, \ldots, 2^n\}$, then number of distinct primes $p \mid M$ is $< n$.
    1. Each prime divisor of $M$ is $\geq 2$, so if $M$ has $t$ distinct prime divisors, then $|M| > 2^t$
    2. $|M| \leq 2^n \Rightarrow t \leq n$
- $F_p(a) \equiv F_p(b)$ if, and only if, $p \mid a - b$.

# Verifying string equality

- If $(a_1, \ldots, a_n) \neq (b_1, \ldots, b_n)$, then $a \neq b$.
- For how many primes can $F_p(a) \equiv F_p(b)$? (i.e., protocol will fail)
- If a number $M$ is in $\{-2^n, \ldots, 2^n\}$, then number of distinct primes $p \mid M$ is $< n$.
  1. Each prime divisor of $M$ is $\geq 2$, so if $M$ has $t$ distinct prime divisors, then $|M| > 2^t$
  2. $|M| \leq 2^n \Rightarrow t \leq n$
- $F_p(a) \equiv F_p(b)$ if, and only if, $p \mid a - b$.
- Thus, protocol fails for at most $n$ choices of $p$

# Verifying string equality

- If $(a_1, \ldots, a_n) \neq (b_1, \ldots, b_n)$, then $a \neq b$.
- For how many primes can $F_p(a) \equiv F_p(b)$? (i.e., protocol will fail)
- If a number $M$ is in $\{-2^n, \ldots, 2^n\}$, then number of distinct primes $p \mid M$ is $< n$.
  1. Each prime divisor of $M$ is $\geq 2$, so if $M$ has $t$ distinct prime divisors, then $|M| > 2^t$
  2. $|M| \leq 2^n \Rightarrow t \leq n$
- $F_p(a) \equiv F_p(b)$ if, and only if, $p \mid a - b$.
- Thus, protocol fails for at most $n$ choices of $p$
- **Prime number theorem**: there are $m/\log m$ primes among first $m$ positive integers

# Verifying string equality

- If $(a_1, \ldots, a_n) \neq (b_1, \ldots, b_n)$, then $a \neq b$.
- For how many primes can $F_p(a) \equiv F_p(b)$? (i.e., protocol will fail)
- If a number $M$ is in $\{-2^n, \ldots, 2^n\}$, then number of distinct primes $p \mid M$ is $< n$.
  1. Each prime divisor of $M$ is $\geq 2$, so if $M$ has $t$ distinct prime divisors, then $|M| > 2^t$
  2. $|M| \leq 2^n \Rightarrow t \leq n$
- $F_p(a) \equiv F_p(b)$ if, and only if, $p \mid a - b$.
- Thus, protocol fails for at most $n$ choices of $p$
- **Prime number theorem**: there are $m/\log m$ primes among first $m$ positive integers
- Choosing $p$ among the first $tn\log(tn)$ primes we have that

$$\Pr[F_p(a) \equiv F_p(b)] \leq \frac{n}{tn\log tn/\log(tn\log tn)} = \tilde{O}(1/t)$$

# Verifying string equality

- If $(a_1, \ldots, a_n) \neq (b_1, \ldots, b_n)$, then $a \neq b$.
- For how many primes can $F_p(a) \equiv F_p(b)$? (i.e., protocol will fail)
- If a number $M$ is in $\{-2^n, \ldots, 2^n\}$, then number of distinct primes $p \mid M$ is $< n$.
  1. Each prime divisor of $M$ is $\geq 2$, so if $M$ has $t$ distinct prime divisors, then $|M| > 2^t$
  2. $|M| \leq 2^n \Rightarrow t \leq n$
- $F_p(a) \equiv F_p(b)$ if, and only if, $p \mid a - b$.
- Thus, protocol fails for at most $n$ choices of $p$
- **Prime number theorem**: there are $m/\log m$ primes among first $m$ positive integers
- Choosing $p$ among the first $tn\log(tn)$ primes we have that

$$\Pr[F_p(a) \equiv F_p(b)] \leq \frac{n}{tn\log tn/\log(tn\log tn)} = \tilde{O}(1/t)$$

- Number of bits sent is $\tilde{O}(\log t + \log n)$. Choosing $t = n$ solves it.

# Polynomial Identity Testing

Technique for string equality testing can be generalized to following setting:

# Polynomial Identity Testing

Technique for string equality testing can be generalized to following setting:

- **Input:** "Given" two polynomials $P(x), Q(x)$, are they equal?
- Two polynomials are equal $\Leftrightarrow$ all their coefficients are equal

# Polynomial Identity Testing

Technique for string equality testing can be generalized to following setting:

- **Input:** "Given" two polynomials $P(x), Q(x)$, are they equal?
- Two polynomials are equal $\Leftrightarrow$ all their coefficients are equal
- So why not just compare their coefficients?

# Polynomial Identity Testing

Technique for string equality testing can be generalized to following setting:

- **Input:** "Given" two polynomials $P(x), Q(x)$, are they equal?
- Two polynomials are equal $\Leftrightarrow$ all their coefficients are equal
- So why not just compare their coefficients?
  1. Sometimes polynomials are given *implicitly* (i.e., not by their list of coefficients)

# Polynomial Identity Testing

Technique for string equality testing can be generalized to following setting:

- **Input:** "Given" two polynomials $P(x), Q(x)$, are they equal?
- Two polynomials are equal $\Leftrightarrow$ all their coefficients are equal
- So why not just compare their coefficients?
  1. Sometimes polynomials are given *implicitly* (i.e., not by their list of coefficients)
  2. $P_1(x), P_2(x), P_3(x)$, test whether: $P_1(x) \cdot P_2(x) = P_3(x)$?

# Polynomial Identity Testing

Technique for string equality testing can be generalized to following setting:

- **Input:** "Given" two polynomials $P(x), Q(x)$, are they equal?
- Two polynomials are equal $\Leftrightarrow$ all their coefficients are equal
- So why not just compare their coefficients?
  1. Sometimes polynomials are given *implicitly* (i.e., not by their list of coefficients)
  2. $P_1(x), P_2(x), P_3(x)$, test whether: $P_1(x) \cdot P_2(x) = P_3(x)$?
  3. If $P_1, P_2$ have degree $\leq n$, then $\deg(P_3) \leq 2n$ (otherwise problem is trivial)

# Polynomial Identity Testing

Technique for string equality testing can be generalized to following setting:

- **Input:** "Given" two polynomials $P(x), Q(x)$, are they equal?
- Two polynomials are equal $\Leftrightarrow$ all their coefficients are equal
- So why not just compare their coefficients?
  1. Sometimes polynomials are given *implicitly* (i.e., not by their list of coefficients)
  2. $P_1(x), P_2(x), P_3(x)$, test whether: $P_1(x) \cdot P_2(x) = P_3(x)$?
  3. If $P_1, P_2$ have degree $\leq n$, then $\deg(P_3) \leq 2n$ (otherwise problem is trivial)
- Multiplication of two polynomials of degree $n$: $O(n \log n)$ arithmetic operations by FFT

# Polynomial Identity Testing

Technique for string equality testing can be generalized to following setting:

- **Input:** "Given" two polynomials $P(x), Q(x)$, are they equal?
- Two polynomials are equal $\Leftrightarrow$ all their coefficients are equal
- So why not just compare their coefficients?
  1. Sometimes polynomials are given *implicitly* (i.e., not by their list of coefficients)
  2. $P_1(x), P_2(x), P_3(x)$, test whether: $P_1(x) \cdot P_2(x) = P_3(x)$?
  3. If $P_1, P_2$ have degree $\leq n$, then $\deg(P_3) \leq 2n$ (otherwise problem is trivial)
- Multiplication of two polynomials of degree $n$: $O(n \log n)$ arithmetic operations by FFT
- Polynomial evaluation: $O(n)$ arithmetic operations

# Polynomial Identity Testing

Technique for string equality testing can be generalized to following setting:

- **Input:** "Given" two polynomials $P(x), Q(x)$, are they equal?
- Two polynomials are equal $\Leftrightarrow$ all their coefficients are equal
- So why not just compute their coefficients?
  1. Sometimes polynomials are given *implicitly* (i.e., not by their list of coefficients)
  2. $P_1(x), P_2(x), P_3(x)$, test whether: $P_1(x) \cdot P_2(x) = P_3(x)$?
  3. If $P_1, P_2$ have degree $\leq n$, then $\deg(P_3) \leq 2n$ (otherwise problem is trivial)
- Multiplication of two polynomials of degree $n$: $O(n \log n)$ arithmetic operations by FFT
- Polynomial evaluation: $O(n)$ arithmetic operations
- Can we check whether $P_1(x) \cdot P_2(x) = P_3(x)$ in $O(n)$ operations?

# Polynomial Identity Testing

Technique for string equality testing can be generalized to following setting:

- **Input:** "Given" two polynomials $P(x), Q(x)$, are they equal?
- Two polynomials are equal $\Leftrightarrow$ all their coefficients are equal
- So why not just compare their coefficients?
  1. Sometimes polynomials are given *implicitly* (i.e., not by their list of coefficients)
  2. $P_1(x), P_2(x), P_3(x)$, test whether: $P_1(x) \cdot P_2(x) = P_3(x)$?
  3. If $P_1, P_2$ have degree $\leq n$, then $\deg(P_3) \leq 2n$ (otherwise problem is trivial)
- Multiplication of two polynomials of degree $n$: $O(n \log n)$ arithmetic operations by FFT
- Polynomial evaluation: $O(n)$ arithmetic operations
- Can we check whether $P_1(x) \cdot P_2(x) = P_3(x)$ in $O(n)$ operations?

**Lemma (Roots of Univariate Polynomials)**

*Let $\mathbb{F}$ be a field and $P(x) \in \mathbb{F}[x]$ be a nonzero univariate polynomial of degree $d$. Then $P(x)$ has at most $d$ distinct roots in $\overline{\mathbb{F}}$.*

# Polynomial Identity Testing

**Lemma (Roots of Univariate Polynomials)**

*Let $\mathbb{F}$ be a field and $P(x) \in \mathbb{F}[x]$ be a nonzero univariate polynomial of degree $d$. Then $P(x)$ has at most $d$ distinct roots in $\overline{\mathbb{F}}$.*

- Let $Q(x) = P_3(x) - P_1(x) \cdot P_2(x)$. It has degree $\leq 2n$

# Polynomial Identity Testing

**Lemma (Roots of Univariate Polynomials)**

*Let $\mathbb{F}$ be a field and $P(x) \in \mathbb{F}[x]$ be a nonzero univariate polynomial of degree $d$. Then $P(x)$ has at most $d$ distinct roots in $\overline{\mathbb{F}}$.*

- Let $Q(x) = P_3(x) - P_1(x) \cdot P_2(x)$. It has degree $\leq 2n$
- By lemma, if $Q \neq 0$ then $Q(a) = 0$ for at most $2n$ values in $\mathbb{F}$.

# Polynomial Identity Testing

**Lemma (Roots of Univariate Polynomials)**

*Let $\mathbb{F}$ be a field and $P(x) \in \mathbb{F}[x]$ be a nonzero univariate polynomial of degree $d$. Then $P(x)$ has at most $d$ distinct roots in $\overline{\mathbb{F}}$.*

- Let $Q(x) = P_3(x) - P_1(x) \cdot P_2(x)$. It has degree $\leq 2n$
- By lemma, if $Q \neq 0$ then $Q(a) = 0$ for at most $2n$ values in $\mathbb{F}$.
- Take a set $S \subseteq \mathbb{F}$ of size $4n$. Let $a \in S$ chosen randomly.

# Polynomial Identity Testing

## Lemma (Roots of Univariate Polynomials)

*Let $\mathbb{F}$ be a field and $P(x) \in \mathbb{F}[x]$ be a nonzero univariate polynomial of degree $d$. Then $P(x)$ has at most $d$ distinct roots in $\overline{\mathbb{F}}$.*

- Let $Q(x) = P_3(x) - P_1(x) \cdot P_2(x)$. It has degree $\leq 2n$
- By lemma, if $Q \neq 0$ then $Q(a) = 0$ for at most $2n$ values in $\mathbb{F}$.
- Take a set $S \subseteq \mathbb{F}$ of size $4n$. Let $a \in S$ chosen randomly.
- Compute $Q(a)$ by computing $P_1(a), P_2(a), P_3(a)$ and then $P_3(a) - P_1(a) \cdot P_2(a)$

# Polynomial Identity Testing

**Lemma (Roots of Univariate Polynomials)**

*Let $\mathbb{F}$ be a field and $P(x) \in \mathbb{F}[x]$ be a nonzero univariate polynomial of degree $d$. Then $P(x)$ has at most $d$ distinct roots in $\overline{\mathbb{F}}$.*

- Let $Q(x) = P_3(x) - P_1(x) \cdot P_2(x)$. It has degree $\leq 2n$
- By lemma, if $Q \neq 0$ then $Q(a) = 0$ for at most $2n$ values in $\mathbb{F}$.
- Take a set $S \subseteq \mathbb{F}$ of size $4n$. Let $a \in S$ chosen randomly.
- Compute $Q(a)$ by computing $P_1(a), P_2(a), P_3(a)$ and then $P_3(a) - P_1(a) \cdot P_2(a)$
- Probability $Q(a) = 0$ (i.e., we failed to identify non-zero)

$$\leq \frac{\deg(Q)}{|S|} \leq \frac{2n}{4n} = 1/2.$$

# Polynomial Identity Testing

**Lemma (Roots of Univariate Polynomials)**

*Let $\mathbb{F}$ be a field and $P(x) \in \mathbb{F}[x]$ be a nonzero univariate polynomial of degree $d$. Then $P(x)$ has at most $d$ distinct roots in $\overline{\mathbb{F}}$.*

- Let $Q(x) = P_3(x) - P_1(x) \cdot P_2(x)$. It has degree $\leq 2n$
- By lemma, if $Q \neq 0$ then $Q(a) = 0$ for at most $2n$ values in $\mathbb{F}$.
- Take a set $S \subseteq \mathbb{F}$ of size $4n$. Let $a \in S$ chosen randomly.
- Compute $Q(a)$ by computing $P_1(a), P_2(a), P_3(a)$ and then $P_3(a) - P_1(a) \cdot P_2(a)$
- Probability $Q(a) = 0$ (i.e., we failed to identify non-zero)

$$\leq \frac{\deg(Q)}{|S|} \leq \frac{2n}{4n} = 1/2.$$

- Can amplify probability by running multiple times or by choosing larger set $S$.

# Polynomial Identity Lemma

**Lemma (Polynomial Identity Lemma)**

*Let $\mathbb{F}$ be a field and $P(x_1, \ldots, x_n) \in \mathbb{F}[x_1, \ldots, x_n]$ be a nonzero polynomial of degree $\leq d$. Then for any set $S \subseteq \overline{\mathbb{F}}$, we have:*

$$\Pr[P(a_1, \ldots, a_n) = 0 \mid a_i \in S] \leq \frac{d}{|S|}$$

# Polynomial Identity Lemma

## Lemma (Polynomial Identity Lemma)

*Let $\mathbb{F}$ be a field and $P(x_1, \ldots, x_n) \in \mathbb{F}[x_1, \ldots, x_n]$ be a nonzero polynomial of degree $\leq d$. Then for any set $S \subseteq \overline{\mathbb{F}}$, we have:*

$$\Pr[P(a_1, \ldots, a_n) = 0 \mid a_i \in S] \leq \frac{d}{|S|}$$

Proof by induction in number of variables.

# Bipartite Matching

- **Input:** bipartite graph $G(L, R, E)$ with $|L| = |R| = n$
- **Output:** does $G$ have a perfect matching?

[2] First proved by Edmonds.

# Bipartite Matching

- **Input:** bipartite graph $G(L, R, E)$ with $|L| = |R| = n$
- **Output:** does $G$ have a perfect matching?
- Let $X \in \mathbb{F}^{n \times n}$ be such that

$$X_{i,j} = \begin{cases} y_{i,j}, & \text{if there is edge between } (i,j) \in L \times R \\ 0, & \text{otherwise} \end{cases}$$

[2] First proved by Edmonds.

# Bipartite Matching

- **Input:** bipartite graph $G(L, R, E)$ with $|L| = |R| = n$
- **Output:** does $G$ have a perfect matching?
- Let $X \in \mathbb{F}^{n \times n}$ be such that

$$X_{i,j} = \begin{cases} y_{i,j}, & \text{if there is edge between } (i,j) \in L \times R \\ 0, & \text{otherwise} \end{cases}$$

- $$\det(X) = \sum_{\sigma \in S_n} (-1)^{\sigma} \prod_{i=1}^{n} X_{i\sigma(i)}$$

[2] First proved by Edmonds.

# Bipartite Matching

- **Input:** bipartite graph $G(L, R, E)$ with $|L| = |R| = n$
- **Output:** does $G$ have a perfect matching?
- Let $X \in \mathbb{F}^{n \times n}$ be such that

$$X_{i,j} = \begin{cases} y_{i,j}, & \text{if there is edge between } (i,j) \in L \times R \\ 0, & \text{otherwise} \end{cases}$$

- $$\det(X) = \sum_{\sigma \in S_n} (-1)^{\sigma} \prod_{i=1}^{n} X_{i\sigma(i)}$$

- $G$ has perfect matching $\Leftrightarrow$ $\det(X)$ is a *non-zero polynomial*![2]

[2] First proved by Edmonds.

# Bipartite Matching

- **Input:** bipartite graph $G(L, R, E)$ with $|L| = |R| = n$
- **Output:** does $G$ have a perfect matching?
- Let $X \in \mathbb{F}^{n \times n}$ be such that

$$X_{i,j} = \begin{cases} y_{i,j}, & \text{if there is edge between } (i,j) \in L \times R \\ 0, & \text{otherwise} \end{cases}$$

- $$\det(X) = \sum_{\sigma \in S_n} (-1)^{\sigma} \prod_{i=1}^{n} X_{i\sigma(i)}$$

- $G$ has perfect matching $\Leftrightarrow \det(X)$ is a *non-zero polynomial*![2]
- Testing if $G$ has a perfect matching is a *special case* of *Polynomial Identity Testing*!

[2] First proved by Edmonds.

# Bipartite Matching

- **Input:** bipartite graph $G(L, R, E)$ with $|L| = |R| = n$
- **Output:** does $G$ have a perfect matching?
- Let $X \in \mathbb{F}^{n \times n}$ be such that

$$X_{i,j} = \begin{cases} y_{i,j}, & \text{if there is edge between } (i,j) \in L \times R \\ 0, & \text{otherwise} \end{cases}$$

- $$\det(X) = \sum_{\sigma \in S_n} (-1)^{\sigma} \prod_{i=1}^{n} X_{i\sigma(i)}$$
- $G$ has perfect matching $\Leftrightarrow$ $\det(X)$ is a *non-zero polynomial*![2]
- Testing if $G$ has a perfect matching is a *special case* of *Polynomial Identity Testing*!
- **Algorithm:** evaluate $\det(X)$ at a random value for the variables $y_{i,j}$.

[2] First proved by Edmonds.

# General Matching

- Ok, bipartite matching is easy (we know many algorithms for it...) what about the general case?

# General Matching

- Ok, bipartite matching is easy (we know many algorithms for it...) what about the general case?
- **Input:** (undirected) graph $G(V, E)$ where $|V| = 2n$.
- **Output:** does $G$ have a perfect matching?

# General Matching

- Ok, bipartite matching is easy (we know many algorithms for it...) what about the general case?
- **Input:** (undirected) graph $G(V, E)$ where $|V| = 2n$.
- **Output:** does $G$ have a perfect matching?
- **Tutte Matrix:** $T_G$ is the following $2n \times 2n$ matrix: let $F$ be an arbitrary orientation of edges in $E$. Then,

$$[T_G]_{i,j} = \begin{cases} x_{i,j} & \text{if } (i,j) \in F \\ -x_{i,j} & \text{if } (j,i) \in F \\ 0 & \text{otherwise} \end{cases}$$

# General Matching

- Ok, bipartite matching is easy (we know many algorithms for it...) what about the general case?
- **Input:** (undirected) graph $G(V, E)$ where $|V| = 2n$.
- **Output:** does $G$ have a perfect matching?
- **Tutte Matrix:** $T_G$ is the following $2n \times 2n$ matrix: let $F$ be an arbitrary orientation of edges in $E$. Then,

$$[T_G]_{i,j} = \begin{cases} x_{i,j} & \text{if } (i,j) \in F \\ -x_{i,j} & \text{if } (j,i) \in F \\ 0 & \text{otherwise} \end{cases}$$

**Theorem (Tutte 1947)**

*$G$ has a perfect matching* $\Leftrightarrow \det(T_G) \neq 0$.

# Where are my parallel algorithms?

We have seen randomized algorithms for bipartite and non-bipartite matching.
Why did you say parallel algorithms?

# Where are my parallel algorithms?

We have seen randomized algorithms for bipartite and non-bipartite matching.
Why did you say parallel algorithms?

- The algorithms for matching consisted of:
  - testing whether a certain determinant is non-zero
  - by evaluating it at a random point

# Where are my parallel algorithms?

We have seen randomized algorithms for bipartite and non-bipartite matching.
Why did you say parallel algorithms?

- The algorithms for matching consisted of:
  - testing whether a certain determinant is non-zero
  - by evaluating it at a random point
- Polynomial Identity Lemma tells us that this algorithm succeeds *with high probability*

# Where are my parallel algorithms?

We have seen randomized algorithms for bipartite and non-bipartite matching.
Why did you say parallel algorithms?

- The algorithms for matching consisted of:
  - testing whether a certain determinant is non-zero
  - by evaluating it at a random point
- Polynomial Identity Lemma tells us that this algorithm succeeds *with high probability*
- We can *compute the determinant efficiently in parallel*

How? See the (future notes) in my CS 860 course.

# Isolation Lemma

Often times in parallel computation, when solving a problem with *many possible solutions*, it is important to make sure that *different processors* are working towards *same solution*.

# Isolation Lemma

Often times in parallel computation, when solving a problem with *many possible solutions*, it is important to make sure that *different processors* are working towards *same solution*.

Need to *single out* (i.e. isolate) a specific solution *without knowing* any element of the solution space. How to do this?

# Isolation Lemma

Often times in parallel computation, when solving a problem with *many possible solutions*, it is important to make sure that *different processors* are working towards *same solution*.

Need to *single out* (i.e. isolate) a specific solution *without knowing* any element of the solution space. How to do this?

- **Solution:** Implicitly choose a *random order* on the feasible solutions and require processors to find solution of *lowest rank* in this order

# Isolation Lemma

Often times in parallel computation, when solving a problem with *many possible solutions*, it is important to make sure that *different processors* are working towards *same solution*.

Need to *single out* (i.e. isolate) a specific solution *without knowing* any element of the solution space. How to do this?

- **Solution:** Implicitly choose a *random order* on the feasible solutions and require processors to find solution of *lowest rank* in this order
- Applications also in distributed computing (breaking deadlocks)!

# Isolation Lemma

Often times in parallel computation, when solving a problem with *many possible solutions*, it is important to make sure that *different processors* are working towards *same solution*.

Need to *single out* (i.e. isolate) a specific solution *without knowing* any element of the solution space. How to do this?

- **Solution:** Implicitly choose a *random order* on the feasible solutions and require processors to find solution of *lowest rank* in this order
- Applications also in distributed computing (breaking deadlocks)!
- Can use it to compute minimum weight perfect matching (see Prof. Lau's notes)

# Isolation Lemma

Often times in parallel computation, when solving a problem with *many possible solutions*, it is important to make sure that *different processors* are working towards *same solution*.

Need to *single out* (i.e. isolate) a specific solution *without knowing* any element of the solution space. How to do this?

- **Solution:** Implicitly choose a *random order* on the feasible solutions and require processors to find solution of *lowest rank* in this order
- Applications also in distributed computing (breaking deadlocks)!
- Can use it to compute minimum weight perfect matching (see Prof. Lau's notes)

**Lemma (Isolation Lemma)**

*Given a set system over* $[n] := \{1, 2, \ldots, n\}$*, if we assign a random weight function* $w : [n] \to [2n]$ *then the probability that there is a unique minimum weight set is at least* $1/2$*.*

# Isolation lemma

## Lemma (Isolation Lemma)

*Given a set system over* $[n] := \{1, 2, \ldots, n\}$*, if we assign a random weight function* $w : [n] \to [2n]$ *then the probability that there is a unique minimum weight set is at least* $1/2$*.*

Example for $n = 4$:

- Set system: $S_1 = \{1, 4\}, S_2 = \{2, 3\}, S_3 = \{1, 2, 3\}$

# Isolation lemma

**Lemma (Isolation Lemma)**

*Given a set system over* $[n] := \{1, 2, \ldots, n\}$*, if we assign a random weight function* $w : [n] \rightarrow [2n]$ *then the probability that there is a unique minimum weight set is at least* $1/2$*.*

Example for $n = 4$:

- Set system: $S_1 = \{1, 4\}, S_2 = \{2, 3\}, S_3 = \{1, 2, 3\}$
- Random weight function $w : [4] \rightarrow [8]$ given by $w(1) = 3, w(2) = 5, w(3) = 8, w(4) = 4$

# Isolation lemma

**Lemma (Isolation Lemma)**

*Given a set system over $[n] := \{1, 2, \ldots, n\}$, if we assign a random weight function $w : [n] \to [2n]$ then the probability that there is a unique minimum weight set is at least $1/2$.*

Example for $n = 4$:

- Set system: $S_1 = \{1, 4\}, S_2 = \{2, 3\}, S_3 = \{1, 2, 3\}$
- Random weight function $w : [4] \to [8]$ given by $w(1) = 3, w(2) = 5, w(3) = 8, w(4) = 4$
- Random weight function $w' : [4] \to [8]$ given by $w'(1) = 5, w'(2) = 1, w'(3) = 7, w'(4) = 3$

# Isolation lemma

**Lemma (Isolation Lemma)**

*Given a set system over* $[n] := \{1, 2, \ldots, n\}$, *if we assign a random weight function* $w : [n] \to [2n]$ *then the probability that there is a unique minimum weight set is at least* $1/2$.

Example for $n = 4$:

- Set system: $S_1 = \{1, 4\}, S_2 = \{2, 3\}, S_3 = \{1, 2, 3\}$
- Random weight function $w : [4] \to [8]$ given by $w(1) = 3, w(2) = 5, w(3) = 8, w(4) = 4$
- Random weight function $w' : [4] \to [8]$ given by $w'(1) = 5, w'(2) = 1, w'(3) = 7, w'(4) = 3$

**Remark**

The isolation lemma could be quite counter-intuitive. A set system can have $\Omega(2^n)$ sets. On average, there are $\Omega(2^n/(2n^2))$ sets of a given weight, as max weight is $\leq 2n^2$. Isolation lemma tells us that with high probability there is *only one* set of minimum weight.

# Proof of Isolation lemma

1. Let $\mathcal{S}$ be our set system and $v \in [n]$.

# Proof of Isolation lemma

1. Let $\mathcal{S}$ be our set system and $v \in [n]$.
2. Let $\mathcal{S}_v$ family of sets from $\mathcal{S}$ which *contain* $v$, and $\mathcal{N}_v$ the family of sets from $\mathcal{S}$ which *do not contain* $v$

# Proof of Isolation lemma

1. Let $\mathcal{S}$ be our set system and $v \in [n]$.
2. Let $\mathcal{S}_v$ family of sets from $\mathcal{S}$ which *contain $v$*, and $\mathcal{N}_v$ the family of sets from $\mathcal{S}$ which *do not contain $v$*
3. Let

$$\alpha_v := \min_{A \in \mathcal{N}_v} w(A) - \min_{B \in \mathcal{S}_v} w(B \setminus \{v\})$$

# Proof of Isolation lemma

1. Let $\mathcal{S}$ be our set system and $v \in [n]$.
2. Let $\mathcal{S}_v$ family of sets from $\mathcal{S}$ which *contain $v$*, and $\mathcal{N}_v$ the family of sets from $\mathcal{S}$ which *do not contain $v$*
3. Let
$$\alpha_v := \min_{A \in \mathcal{N}_v} w(A) - \min_{B \in \mathcal{S}_v} w(B \setminus \{v\})$$
4. $\alpha_v < w(v) \Rightarrow v$ does not belong to any minimum weight set

# Proof of Isolation lemma

1. Let $\mathcal{S}$ be our set system and $v \in [n]$.
2. Let $\mathcal{S}_v$ family of sets from $\mathcal{S}$ which *contain $v$*, and $\mathcal{N}_v$ the family of sets from $\mathcal{S}$ which *do not contain $v$*
3. Let
$$\alpha_v := \min_{A \in \mathcal{N}_v} w(A) - \min_{B \in \mathcal{S}_v} w(B \setminus \{v\})$$
4. $\alpha_v < w(v) \Rightarrow v$ does not belong to any minimum weight set
5. $\alpha_v > w(v) \Rightarrow v$ belongs to every minimum weight set

# Proof of Isolation lemma

1. Let $\mathcal{S}$ be our set system and $v \in [n]$.
2. Let $\mathcal{S}_v$ family of sets from $\mathcal{S}$ which *contain $v$*, and $\mathcal{N}_v$ the family of sets from $\mathcal{S}$ which *do not contain $v$*
3. Let
$$\alpha_v := \min_{A \in \mathcal{N}_v} w(A) - \min_{B \in \mathcal{S}_v} w(B \setminus \{v\})$$
4. $\alpha_v < w(v) \Rightarrow v$ does not belong to any minimum weight set
5. $\alpha_v > w(v) \Rightarrow v$ belongs to every minimum weight set
6. $\alpha_v = w(v) \Rightarrow v$ is ambiguous

# Proof of Isolation lemma

1. Let $\mathcal{S}$ be our set system and $v \in [n]$.
2. Let $\mathcal{S}_v$ family of sets from $\mathcal{S}$ which *contain v*, and $\mathcal{N}_v$ the family of sets from $\mathcal{S}$ which *do not contain v*
3. Let
$$\alpha_v := \min_{A \in \mathcal{N}_v} w(A) - \min_{B \in \mathcal{S}_v} w(B \setminus \{v\})$$
4. $\alpha_v < w(v) \Rightarrow v$ does not belong to any minimum weight set
5. $\alpha_v > w(v) \Rightarrow v$ belongs to every minimum weight set
6. $\alpha_v = w(v) \Rightarrow v$ is ambiguous
7. $\alpha_v$ is *independent* of $w(v)$, and $w(v)$ chosen uniformly at random from $[2n]$.

# Proof of Isolation lemma

3. Let
$$\alpha_v := \min_{A \in \mathcal{N}_v} w(A) - \min_{B \in \mathcal{S}_v} w(B \setminus \{v\})$$
4. $\alpha_v < w(v) \Rightarrow v$ does not belong to any minimum weight set
5. $\alpha_v > w(v) \Rightarrow v$ belongs to every minimum weight set
6. $\alpha_v = w(v) \Rightarrow v$ is ambiguous
7. $\alpha_v$ is *independent* of $w(v)$, and $w(v)$ chosen uniformly at random from $[2n]$.
8. $\Pr[v \text{ ambiguous}] \leq 1/2n \Rightarrow_{\text{union bound}} \Pr[\exists \text{ ambiguous element}] \leq 1/2$

# Proof of Isolation lemma

3. Let
$$\alpha_v := \min_{A \in \mathcal{N}_v} w(A) - \min_{B \in \mathcal{S}_v} w(B \setminus \{v\})$$
4. $\alpha_v < w(v) \Rightarrow v$ does not belong to any minimum weight set
5. $\alpha_v > w(v) \Rightarrow v$ belongs to every minimum weight set
6. $\alpha_v = w(v) \Rightarrow v$ is ambiguous
7. $\alpha_v$ is *independent* of $w(v)$, and $w(v)$ chosen uniformly at random from $[2n]$.
8. $\Pr[v \text{ ambiguous}] \leq 1/2n \Rightarrow_{\text{union bound}} \Pr[\exists \text{ ambiguous element}] \leq 1/2$
9. If two different sets $A, B$ have minimum weight, then any element in $A \Delta B$ must be ambiguous.

# Proof of Isolation lemma

3. Let

$$\alpha_v := \min_{A \in \mathcal{N}_v} w(A) - \min_{B \in \mathcal{S}_v} w(B \setminus \{v\})$$

4. $\alpha_v < w(v) \Rightarrow v$ does not belong to any minimum weight set
5. $\alpha_v > w(v) \Rightarrow v$ belongs to every minimum weight set
6. $\alpha_v = w(v) \Rightarrow v$ is ambiguous
7. $\alpha_v$ is *independent* of $w(v)$, and $w(v)$ chosen uniformly at random from $[2n]$.
8. $\Pr[v \text{ ambiguous}] \leq 1/2n \Rightarrow_{\text{union bound}} \Pr[\exists \text{ ambiguous element}] \leq 1/2$
9. If two different sets $A, B$ have minimum weight, then any element in $A \Delta B$ must be ambiguous.
10. Probability that this happens is $\leq 1/2$. (step 8)

# Potential Final Projects

- Can we derandomize the perfect matching algorithms from class?
- A lot of progress has been made in the past couple years on this question in the works [Fenner, Gurjar & Thierauf 2019] and subsequently [Svensson & Tarnawski 2017]
- Survey of the above, or understanding these papers is a great final project!

# Acknowledgement


- Lecture based largely on:
  - Prof. Lau's notes
  - [Motwani & Raghavan 2007, Chapter 7]
  - [Korte & Vygen 2012, Chapter 10].
- See Prof. Lau's notes at
  `https://cs.uwaterloo.ca/~lapchi/cs466/notes/L07.pdf`

# References I


Motwani, Rajeev and Raghavan, Prabhakar (2007)
Randomized Algorithms

Korte, Bernhard and Vygen, Jens (2012)
Combinatorial optimization. Vol. 2. Heidelberg: Springer.

Fenner, Stephen and Gurjar, Rohit and Thierauf, Thomas (2019)
Bipartite perfect matching is in quasi-NC.
*SIAM Journal on Computing*

Svensson, Ola and Jakub Tarnawski (2017)
The matching problem in general graphs is in quasi-NC.
IEEE 58th Annual Symposium on Foundations of Computer Science

# Proof of Tutte's Theorem

**Theorem (Tutte 1947)**

*G has a perfect matching* $\Leftrightarrow \det(T_G) \neq 0$.

# Proof of Tutte's Theorem

**Theorem (Tutte 1947)**

*G has a perfect matching* $\Leftrightarrow \det(T_G) \neq 0$.

- $$\det(T_G) = \sum_{\sigma \in S_{2n}} (-1)^{\sigma} \prod_{i=1}^{2n} [T_G]_{i,\sigma(i)}$$

# Proof of Tutte's Theorem

**Theorem (Tutte 1947)**

*G has a perfect matching* $\Leftrightarrow \det(T_G) \neq 0$.

- $$\det(T_G) = \sum_{\sigma \in S_{2n}} (-1)^\sigma \prod_{i=1}^{2n} [T_G]_{i,\sigma(i)}$$
- Each permutation $\sigma \in S_{2n}$ that yields non-zero term corresponds to a (directed) subgraph of $G$ $H_\sigma(V, F_\sigma)$, where $F_\sigma = \{(i, \sigma(i))\}_{i=1}^{2n}$.

# Proof of Tutte's Theorem

**Theorem (Tutte 1947)**

*G has a perfect matching* $\Leftrightarrow \det(T_G) \neq 0$.

- $$\det(T_G) = \sum_{\sigma \in S_{2n}} (-1)^{\sigma} \prod_{i=1}^{2n} [T_G]_{i,\sigma(i)}$$
- Each permutation $\sigma \in S_{2n}$ that yields non-zero term corresponds to a (directed) subgraph of $G$ $H_\sigma(V, F_\sigma)$, where $F_\sigma = \{(i, \sigma(i))\}_{i=1}^{2n}$.
- Each vertex in $H_\sigma$ has $|\delta^{out}(i)| = |\delta^{in}(i)| = 1$.

# Proof of Tutte's Theorem

**Theorem (Tutte 1947)**

*G has a perfect matching* $\Leftrightarrow \det(T_G) \neq 0$.

- Each permutation $\sigma \in S_{2n}$ that yields non-zero term corresponds to a (directed) subgraph of $G$ $H_\sigma(V, F_\sigma)$, where $F_\sigma = \{(i, \sigma(i))\}_{i=1}^{2n}$.
- If $\sigma$ only has even cycles, then $H_\sigma$ gives us a perfect matching (by taking every other edge of the graph $H_\sigma$, ignoring orientation)

# Proof of Tutte's Theorem

**Theorem (Tutte 1947)**

*G has a perfect matching* $\Leftrightarrow \det(T_G) \neq 0$.

- Each permutation $\sigma \in S_{2n}$ that yields non-zero term corresponds to a (directed) subgraph of $G$ $H_\sigma(V, F_\sigma)$, where $F_\sigma = \{(i, \sigma(i)\}_{i=1}^{2n}$.
- Otherwise, for each $\sigma \in S_{2n}$ (that has odd cycle), there is another permutation $r(\sigma) \in S_{2n}$ that is obtained by reversing odd cycle of $H_\sigma$ containing vertex with *minimum index*.

# Proof of Tutte's Theorem

**Theorem (Tutte 1947)**

*$G$ has a perfect matching* $\Leftrightarrow \det(T_G) \neq 0$.

# Proof of Tutte's Theorem

**Theorem (Tutte 1947)**

*G has a perfect matching* $\Leftrightarrow \det(T_G) \neq 0$.

- Comparing $(-1)^{\sigma} \prod_{i=1}^{2n} [T_G]_{i,\sigma(i)}$ and $(-1)^{r(\sigma)} \prod_{i=1}^{2n} [T_G]_{i,r(\sigma)(i)}$

# Proof of Tutte's Theorem

**Theorem (Tutte 1947)**

*G has a perfect matching* $\Leftrightarrow \det(T_G) \neq 0.$

- Comparing $(-1)^{\sigma} \prod_{i=1}^{2n} [T_G]_{i,\sigma(i)}$ and $(-1)^{r(\sigma)} \prod_{i=1}^{2n} [T_G]_{i,r(\sigma)(i)}$
  - $(-1)^{\sigma} = (-1)^{r(\sigma)} \Leftarrow$ cycles of same size

# Proof of Tutte's Theorem

**Theorem (Tutte 1947)**

*G has a perfect matching* $\Leftrightarrow \det(T_G) \neq 0.$

- Comparing $(-1)^{\sigma} \prod_{i=1}^{2n} [T_G]_{i,\sigma(i)}$ and $(-1)^{r(\sigma)} \prod_{i=1}^{2n} [T_G]_{i,r(\sigma)(i)}$
  - $(-1)^{\sigma} = (-1)^{r(\sigma)} \Leftarrow$ cycles of same size
  - $$\prod_{i=1}^{2n} [T_G]_{i,\sigma(i)} = \prod_{i=1}^{2n} x_{i,\sigma(i)} = -\prod_{i=1}^{2n} [T_G]_{i,r(\sigma)(i)}$$

# Proof of Tutte's Theorem

## Theorem (Tutte 1947)

*$G$ has a perfect matching* $\Leftrightarrow \det(T_G) \neq 0$.

- Comparing $(-1)^{\sigma} \prod_{i=1}^{2n} [T_G]_{i,\sigma(i)}$ and $(-1)^{r(\sigma)} \prod_{i=1}^{2n} [T_G]_{i,r(\sigma)(i)}$
  - $(-1)^{\sigma} = (-1)^{r(\sigma)} \Leftarrow$ cycles of same size
  - $$\prod_{i=1}^{2n} [T_G]_{i,\sigma(i)} = \prod_{i=1}^{2n} x_{i,\sigma(i)} = -\prod_{i=1}^{2n} [T_G]_{i,r(\sigma)(i)}$$
  - These two terms *cancel*!

# Proof of Tutte's Theorem

**Theorem (Tutte 1947)**

*G has a perfect matching* $\Leftrightarrow \det(T_G) \neq 0$.

- Comparing $(-1)^{\sigma} \prod_{i=1}^{2n} [T_G]_{i,\sigma(i)}$ and $(-1)^{r(\sigma)} \prod_{i=1}^{2n} [T_G]_{i,r(\sigma)(i)}$
    - $(-1)^{\sigma} = (-1)^{r(\sigma)} \Leftarrow$ cycles of same size
    - $$\prod_{i=1}^{2n} [T_G]_{i,\sigma(i)} = \prod_{i=1}^{2n} x_{i,\sigma(i)} = -\prod_{i=1}^{2n} [T_G]_{i,r(\sigma)(i)}$$
    - These two terms *cancel*!
- Since $r(r(\sigma)) = \sigma$, all such terms cancel!

# Proof of Tutte's Theorem

**Theorem (Tutte 1947)**

*G has a perfect matching* $\Leftrightarrow \det(T_G) \neq 0$.

- Comparing $(-1)^{\sigma} \prod_{i=1}^{2n} [T_G]_{i,\sigma(i)}$ and $(-1)^{r(\sigma)} \prod_{i=1}^{2n} [T_G]_{i,r(\sigma)(i)}$
  - $(-1)^{\sigma} = (-1)^{r(\sigma)} \Leftarrow$ cycles of same size
  - $$\prod_{i=1}^{2n} [T_G]_{i,\sigma(i)} = \prod_{i=1}^{2n} x_{i,\sigma(i)} = -\prod_{i=1}^{2n} [T_G]_{i,r(\sigma)(i)}$$
  - These two terms *cancel*!
- Since $r(r(\sigma)) = \sigma$, all such terms cancel!
- Is there a term that does not cancel? (have to show that $\det(T_G) \not\equiv 0$)

# Proof of Tutte's Theorem

**Theorem (Tutte 1947)**

*G has a perfect matching* $\Leftrightarrow \det(T_G) \neq 0$.

- Is there a term that does not cancel? (have to show that $\det(T_G) \not\equiv 0$)
- If $T_G$ has a matching, say, $\{1,2\},\{3,4\},\ldots,\{2n-1,2n\}$, then take permutation $\sigma = (1\ 2)(3\ 4)\cdots(2n-1\ 2n)$

$$(-1)^{\sigma}\prod_{i=1}^{2n}[T_G]_{i,\sigma(i)} = (-1)^n \prod_{i=1}^{n} -x^2_{(2i-1)\sigma(2i-1)} = \prod_{i=1}^{n} x^2_{(2i-1)\sigma(2i-1)}.$$