New results on Noncommutative and Commutative Polynomial Identity Testing

V. Arvind, Partha Mukhopadhyay, and Srikanth Srinivasan
The Institute of Mathematical Sciences
India

25th June 2008

Arithmetic Circuit

Definition

An arithmetic circuit over a field $\mathbb{F}$ is a circuit with addition and multiplication gates. The inputs to a gate is either variables, constants from $\mathbb{F}$ or outputs of other gates. An arithmetic circuit C with the inputs $x_1, x_2, \cdots, x_n$ computes a polynomial in $\mathbb{F}[x_1, x_2, \cdots, x_n]$.

Polynomial Identity Testing Problem

Definition

Let $\mathbb{F}$ be a field and C be an arithmetic circuit in the input variable $x_1, x_2, \cdots, x_n$ over $\mathbb{F}$. Can one determine whether the polynomial computed by C is identically zero ?

History of the problem

- It is a well known classical problem.
- Randomized polynomial time algorithm is known (Schwartz-Zippel 1978).
- No deterministic polynomial time algorithm is known.
- Impagliazzo and Kabanets (2003) showed that such an algorithm will imply either $\mathrm{NEXP} \not\subset \mathrm{P/poly}$ or Permanent has no polynomial size arithmetic circuit.

History of the problem

- It is a well known classical problem.
- Randomized polynomial time algorithm is known (Schwartz-Zippel 1978).
- No deterministic polynomial time algorithm is known.
- Impagliazzo and Kabanets (2003) showed that such an algorithm will imply either $\mathrm{NEXP} \not\subset \mathrm{P/poly}$ or Permanent has no polynomial size arithmetic circuit.

History of the problem

- It is a well known classical problem.
- Randomized polynomial time algorithm is known (Schwartz-Zippel 1978).
- No deterministic polynomial time algorithm is known.
- Impagliazzo and Kabanets (2003) showed that such an algorithm will imply either $\mathrm{NEXP} \not\subset \mathrm{P/poly}$ or Permanent has no polynomial size arithmetic circuit.

History of the problem

- It is a well known classical problem.
- Randomized polynomial time algorithm is known (Schwartz-Zippel 1978).
- No deterministic polynomial time algorithm is known.
- Impagliazzo and Kabanets (2003) showed that such an algorithm will imply either $\mathrm{NEXP} \not\subset \mathrm{P/poly}$ or Permanent has no polynomial size arithmetic circuit.

Noncommutative Model of computation

- In this talk we are primarily interested in noncommutative model, where the input variables x_i, x_j do not commute, i.e $x_i x_j - x_j x_i \neq 0$.
- The output of the arithmetic circuit C is a formal expression in the noncommutative ring $\mathbb{F}\{x_1, x_2, \cdots, x_n\}$.
- Problem is to test whether C computes an identically zero expression.

Noncommutative Model of computation

- In this talk we are primarily interested in noncommutative model, where the input variables x_i, x_j do not commute, i.e $x_i x_j - x_j x_i \neq 0$.
- The output of the arithmetic circuit C is a formal expression in the noncommutative ring $\mathbb{F}\{x_1, x_2, \cdots, x_n\}$.
- Problem is to test whether C computes an identically zero expression.

Noncommutative Model of computation

- In this talk we are primarily interested in noncommutative model, where the input variables x_i, x_j do not commute, i.e $x_i x_j - x_j x_i \neq 0$.
- The output of the arithmetic circuit C is a formal expression in the noncommutative ring $\mathbb{F}\{x_1, x_2, \cdots, x_n\}$.
- Problem is to test whether C computes an identically zero expression.

Known results over Noncommutative model

Identity Testing Results

- Raz and Shpilka (2005) designed deterministic polynomial time algorithm for noncommutative formula.
- Bogdanov and Wee (2005) showed a randomized polynomial time identity testing algorithm for circuit computing polynomial of small degree.

Known results over Noncommutative model

Identity Testing Results

- Raz and Shpilka (2005) designed deterministic polynomial time algorithm for noncommutative formula.
- Bogdanov and Wee (2005) showed a randomized polynomial time identity testing algorithm for circuit computing polynomial of small degree.

Known results over Noncommutative model

Lower Bounds

- Nisan (1991) showed exponential size lower bounds for noncommutative formulas that compute the noncommutative permanent or determinant polynomials.
- Chien and Sinclair (2004) extended Nisan's results over different algebras.

Known results over Noncommutative model

Lower Bounds

- Nisan (1991) showed exponential size lower bounds for noncommutative formulas that compute the noncommutative permanent or determinant polynomials.
- Chien and Sinclair (2004) extended Nisan's results over different algebras.

Our Main Results

- Given a noncommutative circuit computing a sparse polynomial of small degree, we give a deterministic polynomial-time identity testing algorithm.
- Given a noncommutative circuit computing a sparse polynomial of small degree, we give a deterministic polynomial-time algorithm to reconstruct the entire polynomial. (In the commutative case, Ben-Or and Tiwari (1988) showed a deterministic polynomial time interpolation algorithm for sparse multivariate polynomial)

Our Main Results

- Given a noncommutative circuit computing a sparse polynomial of small degree, we give a deterministic polynomial-time identity testing algorithm.
- Given a noncommutative circuit computing a sparse polynomial of small degree, we give a deterministic polynomial-time algorithm to reconstruct the entire polynomial. (In the commutative case, Ben-Or and Tiwari (1988) showed a deterministic polynomial time interpolation algorithm for sparse multivariate polynomial)

Our Main Results

- In a suitably defined black-box model, we show an efficient reconstruction algorithm for noncommuting Algebraic Branching Program (ABP).

Automata Theory Background

Building blocks of our algorithm

- A finite automaton $A = (Q, \Sigma, \delta, q_0, q_f)$.
  - Input alphabet $\Sigma = \{0, 1\}$.
  - Q is the set of states.
  - $\delta : Q \times \{0, 1\} \to Q$ is the transition function.
  - q_0 and q_f are the initial and final states.
- For $b \in \{0, 1\}$, define the 0-1 matrix $M_b \in \mathbb{F}^{|Q| \times |Q|}$:

$$M_b(q, q') = \begin{cases} 1 & \text{if } \delta_b(q) = q', \\ 0 & \text{otherwise.} \end{cases}$$

Automata Theory Background

Building blocks of our algorithm

- A finite automaton $A = (Q, \Sigma, \delta, q_0, q_f)$.
  - Input alphabet $\Sigma = \{0, 1\}$.
  - Q is the set of states.
  - $\delta : Q \times \{0, 1\} \to Q$ is the transition function.
  - q_0 and q_f are the initial and final states.
- For $b \in \{0, 1\}$, define the 0-1 matrix $M_b \in \mathbb{F}^{|Q|\times|Q|}$:

$$M_b(q, q') = \begin{cases} 1 & \text{if } \delta_b(q) = q', \\ 0 & \text{otherwise.} \end{cases}$$

Automata Theory Background

Building blocks of our algorithm

- A finite automaton $A = (Q, \Sigma, \delta, q_0, q_f)$.
  - Input alphabet $\Sigma = \{0, 1\}$.
  - Q is the set of states.
  - $\delta : Q \times \{0, 1\} \to Q$ is the transition function.
  - q_0 and q_f are the initial and final states.
- For $b \in \{0, 1\}$, define the 0-1 matrix $M_b \in \mathbb{F}^{|Q| \times |Q|}$:

$$M_b(q, q') = \begin{cases} 1 & \text{if } \delta_b(q) = q', \\ 0 & \text{otherwise.} \end{cases}$$

Automata Theory Background

Building blocks of our algorithm

- A finite automaton $A = (Q, \Sigma, \delta, q_0, q_f)$.
  - Input alphabet $\Sigma = \{0, 1\}$.
  - Q is the set of states.
  - $\delta : Q \times \{0, 1\} \to Q$ is the transition function.
  - q_0 and q_f are the initial and final states.
- For $b \in \{0, 1\}$, define the 0-1 matrix $M_b \in \mathbb{F}^{|Q| \times |Q|}$:

$$M_b(q, q') = \begin{cases} 1 & \text{if } \delta_b(q) = q', \\ 0 & \text{otherwise.} \end{cases}$$

Automata Theory Background

Building blocks of our algorithm

- A finite automaton $A = (Q, \Sigma, \delta, q_0, q_f)$.
  - Input alphabet $\Sigma = \{0, 1\}$.
  - Q is the set of states.
  - $\delta : Q \times \{0, 1\} \rightarrow Q$ is the transition function.
  - q_0 and q_f are the initial and final states.
- For $b \in \{0, 1\}$, define the 0-1 matrix $M_b \in \mathbb{F}^{|Q|\times|Q|}$:

$$M_b(q, q') = \begin{cases} 1 & \text{if } \delta_b(q) = q', \\ 0 & \text{otherwise.} \end{cases}$$

Automata Theory Background

Building blocks of our algorithm

- A finite automaton $A = (Q, \Sigma, \delta, q_0, q_f)$.
  - Input alphabet $\Sigma = \{0, 1\}$.
  - Q is the set of states.
  - $\delta : Q \times \{0, 1\} \to Q$ is the transition function.
  - q_0 and q_f are the initial and final states.
- For $b \in \{0, 1\}$, define the 0-1 matrix $M_b \in \mathbb{F}^{|Q| \times |Q|}$:

$$M_b(q, q') = \begin{cases} 1 & \text{if } \delta_b(q) = q', \\ 0 & \text{otherwise.} \end{cases}$$

Automata Theory Background

Building blocks of our algorithm

- For any $w = w_1 w_2 \cdots w_k \in \{0,1\}^*$, the matrix $M_w = M_{w_1} M_{w_2} \cdots M_{w_k}$.
- Easy fact:

$$M_w(q, q') = \begin{cases} 1 & \text{if } \delta_w(q) = q', \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

- $M_w(q_0, q_f) = 1$ if and only if w is accepted by the automaton A.

Automata Theory Background

Building blocks of our algorithm

- For any $w = w_1 w_2 \cdots w_k \in \{0,1\}^*$, the matrix $M_w = M_{w_1} M_{w_2} \cdots M_{w_k}$.
- Easy fact:

$$M_w(q,q') = \begin{cases} 1 & \text{if } \delta_w(q) = q', \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

- $M_w(q_0, q_f) = 1$ if and only if w is accepted by the automaton A.

Automata Theory Background

Building blocks of our algorithm

- For any $w = w_1 w_2 \cdots w_k \in \{0,1\}^*$, the matrix $M_w = M_{w_1} M_{w_2} \cdots M_{w_k}$.
- Easy fact:

$$M_w(q, q') = \begin{cases} 1 & \text{if } \delta_w(q) = q', \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

- $M_w(q_0, q_f) = 1$ if and only if w is accepted by the automaton A.

Run of an automaton over a noncommutative circuit

- Encode the variable x_i in the alphabet $\{0,1\}$ by the string $v_i = 01^i0$.
- For given automaton A, the matrix $M_{v_i} = M_0 M_1^i M_0$.
- Let C be the given arithmetic circuit computing a polynomial f in $\mathbb{F}\{x_1, x_2, \cdots, x_n\}$.
- Compute the output matrix $M_{out}^A = C(M_{v_1}, M_{v_2} \cdots, M_{v_n})$.

Run of an automaton over a noncommutative circuit

- Encode the variable x_i in the alphabet $\{0, 1\}$ by the string $v_i = 01^i0$.
- For given automaton A, the matrix $M_{v_i} = M_0 M_1^i M_0$.
- Let C be the given arithmetic circuit computing a polynomial f in $\mathbb{F}\{x_1, x_2, \cdots, x_n\}$.
- Compute the output matrix $M_{out}^A = C(M_{v_1}, M_{v_2} \cdots, M_{v_n})$.

Run of an automaton over a noncommutative circuit

- Encode the variable x_i in the alphabet $\{0,1\}$ by the string $v_i = 01^i0$.
- For given automaton A, the matrix $M_{v_i} = M_0 M_1^i M_0$.
- Let C be the given arithmetic circuit computing a polynomial f in $\mathbb{F}\{x_1, x_2, \cdots, x_n\}$.
- Compute the output matrix $M_{out}^A = C(M_{v_1}, M_{v_2} \cdots, M_{v_n})$.

Run of an automaton over a noncommutative circuit

- Encode the variable x_i in the alphabet $\{0, 1\}$ by the string $v_i = 01^i0$.
- For given automaton A, the matrix $M_{v_i} = M_0 M_1^i M_0$.
- Let C be the given arithmetic circuit computing a polynomial f in $\mathbb{F}\{x_1, x_2, \cdots, x_n\}$.
- Compute the output matrix $M_{out}^A = C(M_{v_1}, M_{v_2} \cdots, M_{v_n})$.

Crucial Observation

- f determines M^A_{out} completely; the structure C is otherwise irrelevant.
- The output is always 0 when $f \equiv 0$.
- If $f(x_1, \cdots, x_n) = cx_{j_1} \cdots x_{j_k}$, with $c \in \mathbb{F}$, then $M^A_{out} = cM_{v_{j_1}} \cdots M_{v_{j_k}}$ where $x_{j_i} \rightarrow v_{j_i} = 01^{j_i}1$.

Crucial Observation

- f determines M^A_{out} completely; the structure C is otherwise irrelevant.
- The output is always 0 when $f \equiv 0$.
- If $f(x_1, \cdots, x_n) = cx_{j_1} \cdots x_{j_k}$, with $c \in \mathbb{F}$, then $M^A_{out} = cM_{v_{j_1}} \cdots M_{v_{j_k}}$ where $x_{j_i} \to v_{j_i} = 01^{j_i}1$.

Crucial Observation

- f determines M^A_{out} completely; the structure C is otherwise irrelevant.
- The output is always 0 when $f \equiv 0$.
- If $f(x_1, \cdots, x_n) = cx_{j_1} \cdots x_{j_k}$, with $c \in \mathbb{F}$, then $M^A_{out} = cM_{v_{j_1}} \cdots M_{v_{j_k}}$ where $x_{j_i} \to v_{j_i} = 01^{j_i}1$.

Crucial Observation

- The entry $M_{out}^{A}(q_0, q_f)$ is 0 when A rejects $m = x_{j_1} \cdots x_{j_k}$ (i.e it's binary representation), and c when A accepts m.
- In general, let $f = \sum_i c_i m_i$, then $M_{out}^{A}(q_0, q_f) = \sum_j c_j$ such that m_j's are accepted by A.

Crucial Observation

- The entry $M^A_{out}(q_0, q_f)$ is 0 when A rejects $m = x_{j_1} \cdots x_{j_k}$ (i.e it's binary representation), and c when A accepts m.
- In general, let $f = \sum_i c_i m_i$, then $M^A_{out}(q_0, q_f) = \sum_j c_j$ such that m_j's are accepted by A.

Intuition for Identity Testing

- Can one design a small-sized automaton A such that A accepts precisely one monomial m (with coefficient c) of the polynomial computed by C.
- Looking at (q_0, q_f) entry of M^A_{out} (which is c), we can confirm that $f \not\equiv 0$.
- Such an automaton A is a *good automaton* for us.
- Even designing a small family of automata with a guarantee that the family contains a *good automaton* is enough.

Intuition for Identity Testing

- Can one design a small-sized automaton A such that A accepts precisely one monomial m (with coefficient c) of the polynomial computed by C.
- Looking at (q_0, q_f) entry of M_{out}^A (which is c), we can confirm that $f \not\equiv 0$.
- Such an automaton A is a *good automaton* for us.
- Even designing a small family of automata with a guarantee that the family contains a *good automaton* is enough.

Intuition for Identity Testing

- Can one design a small-sized automaton A such that A accepts precisely one monomial m (with coefficient c) of the polynomial computed by C.
- Looking at (q_0, q_f) entry of M^A_{out} (which is c), we can confirm that $f \not\equiv 0$.
- Such an automaton A is a *good automaton* for us.
- Even designing a small family of automata with a guarantee that the family contains a *good automaton* is enough.

Intuition for Identity Testing

- Can one design a small-sized automaton A such that A accepts precisely one monomial m (with coefficient c) of the polynomial computed by C.
- Looking at (q_0, q_f) entry of M^A_{out} (which is c), we can confirm that $f \not\equiv 0$.
- Such an automaton A is a *good automaton* for us.
- Even designing a small family of automata with a guarantee that the family contains a *good automaton* is enough.

An isolating family of finite automata

- Let W be any finite set of at most s binary strings of length at most m.
- Let $\mathcal{A}$ be a finite family of finite automata over the binary alphabet $\{0, 1\}$.
- $\mathcal{A}$ is a (m, s)-*isolating family* for W, if there is a $A \in \mathcal{A}$ such that A accepts precisely one string from W.

An isolating family of finite automata

- Let W be any finite set of at most s binary strings of length at most m.
- Let $\mathcal{A}$ be a finite family of finite automata over the binary alphabet $\{0, 1\}$.
- $\mathcal{A}$ is a (m, s)-*isolating family* for W, if there is a $A \in \mathcal{A}$ such that A accepts precisely one string from W.

An isolating family of finite automata

- Let W be any finite set of at most s binary strings of length at most m.
- Let $\mathcal{A}$ be a finite family of finite automata over the binary alphabet $\{0, 1\}$.
- $\mathcal{A}$ is a (m, s)-*isolating family* for W, if there is a $A \in \mathcal{A}$ such that A accepts precisely one string from W.

Identity Testing Algorithm

- C be a given arithmetic circuit computing a polynomial $f \in \mathbb{F}\{x_1, x_2, \cdots, x_n\}$ of degree at most d and number of monomials is at most t.
- Monomials of f correspond to binary strings of length at most $d(n+2)$.
- So it is enough to construct a universal family of automata $\mathcal{A}$ which is a $(d(n+2), t)$-isolating family.
- For identity testing we just need to run the automata $A \in \mathcal{A}$ over C and look into the (q_0, q_f) entry of M^A_{out}.

Identity Testing Algorithm

- C be a given arithmetic circuit computing a polynomial $f \in \mathbb{F}\{x_1, x_2, \cdots, x_n\}$ of degree at most d and number of monomials is at most t.
- Monomials of f correspond to binary strings of length at most $d(n+2)$.
- So it is enough to construct a universal family of automata $\mathcal{A}$ which is a $(d(n+2), t)$-isolating family.
- For identity testing we just need to run the automata $A \in \mathcal{A}$ over C and look into the (q_0, q_f) entry of M^A_{out}.

Identity Testing Algorithm

- C be a given arithmetic circuit computing a polynomial $f \in \mathbb{F}\{x_1, x_2, \cdots, x_n\}$ of degree at most d and number of monomials is at most t.
- Monomials of f correspond to binary strings of length at most $d(n+2)$.
- So it is enough to construct a universal family of automata $\mathcal{A}$ which is a $(d(n+2), t)$-isolating family.
- For identity testing we just need to run the automata $A \in \mathcal{A}$ over C and look into the (q_0, q_f) entry of M_{out}^A.

Identity Testing Algorithm

- C be a given arithmetic circuit computing a polynomial $f \in \mathbb{F}\{x_1, x_2, \cdots, x_n\}$ of degree at most d and number of monomials is at most t.
- Monomials of f correspond to binary strings of length at most $d(n+2)$.
- So it is enough to construct a universal family of automata $\mathcal{A}$ which is a $(d(n+2), t)$-isolating family.
- For identity testing we just need to run the automata $A \in \mathcal{A}$ over C and look into the (q_0, q_f) entry of M^A_{out}.

Construction of an isolating automata family

- W be a set of s binary strings each of length at most m. Our goal is to construct a (m, s)-isolating automata family.
- For a string $w \in \{0, 1\}^*$, let n_w be the positive integer represented by the binary numeral $1w$.
- For a prime p and an integer $i \in \{0, \cdots, p-1\}$, construct an automaton $A_{p,i}$ (having exactly one accepting state) that accepts exactly those w such that $n_w \equiv i \pmod{p}$.

Construction of an isolating automata family

- W be a set of s binary strings each of length at most m. Our goal is to construct a (m, s)-isolating automata family.
- For a string $w \in \{0,1\}^*$, let n_w be the positive integer represented by the binary numeral $1w$.
- For a prime p and an integer $i \in \{0, \cdots, p-1\}$, construct an automaton $A_{p,i}$ (having exactly one accepting state) that accepts exactly those w such that $n_w \equiv i \pmod{p}$.

Construction of an isolating automata family

- W be a set of s binary strings each of length at most m. Our goal is to construct a (m, s)-isolating automata family.
- For a string $w \in \{0,1\}^*$, let n_w be the positive integer represented by the binary numeral $1w$.
- For a prime p and an integer $i \in \{0, \cdots, p-1\}$, construct an automaton $A_{p,i}$ (having exactly one accepting state) that accepts exactly those w such that $n_w \equiv i \pmod{p}$.

Construction of an isolating automata family

- $A_{p,i}$ isolates W if there exists j such that $n_{w_j} - n_{w_k} \not\equiv 0 \pmod{p}$ for $k \neq j$ and $n_{w_j} \equiv i \pmod{p}$.
- So to construct an isolating family it is enough to avoid prime factors of $P = \prod_{j \neq k}(n_{w_j} - n_{w_k})$.
- The number of prime factors of P is clearly bounded by $(m+2)\binom{s}{2}$.

Construction of an isolating automata family

- $A_{p,i}$ isolates W if there exists j such that $n_{w_j} - n_{w_k} \not\equiv 0 \pmod p$ for $k \neq j$ and $n_{w_j} \equiv i \pmod p$.
- So to construct an isolating family it is enough to avoid prime factors of $P = \prod_{j \neq k}(n_{w_j} - n_{w_k})$.
- The number of prime factors of P is clearly bounded by $(m+2)\binom{s}{2}$.

Construction of an isolating automata family

- $A_{p,i}$ isolates W if there exists j such that $n_{w_j} - n_{w_k} \not\equiv 0 \pmod{p}$ for $k \neq j$ and $n_{w_j} \equiv i \pmod{p}$.
- So to construct an isolating family it is enough to avoid prime factors of $P = \prod_{j \neq k}(n_{w_j} - n_{w_k})$.
- The number of prime factors of P is clearly bounded by $(m+2)\binom{s}{2}$.

Construction of isolating family continued

- Consider $N = (m+2)\binom{s}{2} + 1$.
- Isolating automata family: $\{A_{p,i}\}_{p,i}$ where p runs over the first N primes, and $i \in \{0, 1, \cdots, p-1\}$.

Construction of isolating family continued

- Consider $N = (m+2)\binom{s}{2} + 1$.
- Isolating automata family: $\{A_{p,i}\}_{p,i}$ where p runs over the first N primes, and $i \in \{0, 1, \cdots, p-1\}$.

The Interpolation Algorithm

- Input: An arithmetic circuit C computing a polynomial $f \in \mathbb{F}\{x_1, x_2, \cdots, x_n\}$. Let d and t are the upper bounds on the degree and number of monomials of f.
- Goal: To compute the polynomial f explicitly in time $\mathrm{poly}(|C|, n, d, t)$.
- Idea: Prefix search based recursive algorithm.

The Interpolation Algorithm

- Input: An arithmetic circuit C computing a polynomial $f \in \mathbb{F}\{x_1, x_2, \cdots, x_n\}$. Let d and t are the upper bounds on the degree and number of monomials of f.
- Goal: To compute the polynomial f explicitly in time $\mathrm{poly}(|C|, n, d, t)$.
- Idea: Prefix search based recursive algorithm.

The Interpolation Algorithm

- Input: An arithmetic circuit C computing a polynomial $f \in \mathbb{F}\{x_1, x_2, \cdots, x_n\}$. Let d and t are the upper bounds on the degree and number of monomials of f.
- Goal: To compute the polynomial f explicitly in time $\mathrm{poly}(|C|, n, d, t)$.
- Idea: Prefix search based recursive algorithm.

Prefix search based recursion

- Given C and a monomial u, $\mathtt{Interpolate}(\mathtt{C}, \mathtt{u})$ finds all the monomials of f (along with their coefficients) which contain u as prefix. So to compute entire polynomial we invoke $\mathtt{Interpolate}(\mathtt{C}, \epsilon)$.

Some Notations

- For a string u (think of as encoded in binary), A_u is the standard automaton that accepts only u.
- For an automaton A, let $[A]_u$ is the automaton that accepts precisely those strings accepted by A which contain u as a prefix.
- For a family of automata $\mathcal{A}$, $[\mathcal{A}]_u = \{[A]_u \mid A \in \mathcal{A}\}$.

Some Notations

- For a string u (think of as encoded in binary), A_u is the standard automaton that accepts only u.
- For an automaton A, let $[A]_u$ is the automaton that accepts precisely those strings accepted by A which contain u as a prefix.
- For a family of automata $\mathcal{A}$, $[\mathcal{A}]_u = \{[A]_u \mid A \in \mathcal{A}\}$.

Some Notations

- For a string u (think of as encoded in binary), A_u is the standard automaton that accepts only u.
- For an automaton A, let $[A]_u$ is the automaton that accepts precisely those strings accepted by A which contain u as a prefix.
- For a family of automata $\mathcal{A}$, $[\mathcal{A}]_u = \{[A]_u \mid A \in \mathcal{A}\}$.

Isolating Automata Family

- Fix a (m, s)-Isolating automata family $\mathcal{A}$, with $m = d(n+2)$ and $s = t$.
- There exists a *good prime* p such that for every monomial w of f the following is true: There exists $i \in [p-1]$, such that $A_{p,i} \in \mathcal{A}$ accepts w (i.e it's binary representation) and rejects all other monomials of f.

Isolating Automata Family

- Fix a (m, s)-Isolating automata family $\mathcal{A}$, with $m = d(n+2)$ and $s = t$.
- There exists a *good prime* p such that for every monomial w of f the following is true: There exists $i \in [p-1]$, such that $A_{p,i} \in \mathcal{A}$ accepts w (i.e it's binary representation) and rejects all other monomials of f.

Building blocks of the Interpolation Algorithm

- Given a monomial u, it is easy to check whether u is a nonzero monomial in f: Compute the run of A_u on C. The (q_o, q_f) entry of $M_{out}^{A_u}$ is the coefficient of u in f.
- If u is the prefix of some monomial v in f, some automaton in $A \in [\mathcal{A}]_u$ will accept u.
- To check whether u appears as a prefix of any monomial in f: Compute the run of $A \in [\mathcal{A}]_u$ on C. Check whether the (q_0, q_f) entry of M_{out}^{A} is nonzero for some A.

Building blocks of the Interpolation Algorithm

- Given a monomial u, it is easy to check whether u is a nonzero monomial in f: Compute the run of A_u on C. The (q_o, q_f) entry of $M_{out}^{A_u}$ is the coefficient of u in f.
- If u is the prefix of some monomial v in f, some automaton in $A \in [\mathcal{A}]_u$ will accept u.
- To check whether u appears as a prefix of any monomial in f: Compute the run of $A \in [\mathcal{A}]_u$ on C. Check whether the (q_0, q_f) entry of M_{out}^{A} is nonzero for some A.

Building blocks of the Interpolation Algorithm

- Given a monomial u, it is easy to check whether u is a nonzero monomial in f: Compute the run of A_u on C. The (q_o, q_f) entry of $M_{out}^{A_u}$ is the coefficient of u in f.
- If u is the prefix of some monomial v in f, some automaton in $A \in [\mathcal{A}]_u$ will accept u.
- To check whether u appears as a prefix of any monomial in f: Compute the run of $A \in [\mathcal{A}]_u$ on C. Check whether the (q_0, q_f) entry of M_{out}^{A} is nonzero for some A.

Interpolation Algorithm

`Interpolate(`C`,`u`)`

- Compute the coefficient of u in f.
- Check whether $u0$ is a prefix of any monomial in f. If so, `Interpolate(`C`,`$u0$`)`.
- Check whether $u1$ is a prefix of any monomial in f. If so, `Interpolate(`C`,`$u1$`)`.

Interpolation Algorithm

`Interpolate(`C, u`)`

- Compute the coefficient of u in f.
- Check whether $u0$ is a prefix of any monomial in f. If so, `Interpolate(`C,$u0$`)`.
- Check whether $u1$ is a prefix of any monomial in f. If so, `Interpolate(`C,$u1$`)`.

Interpolation Algorithm

`Interpolate`(C, u)

- Compute the coefficient of u in f.
- Check whether $u0$ is a prefix of any monomial in f. If so, `Interpolate`$(C, u0)$.
- Check whether $u1$ is a prefix of any monomial in f. If so, `Interpolate`$(C, u1)$.

Running time of the algorithm

- The algorithm calls `Interpolate` on u only if u is the prefix of some string corresponding to a monomial in f.
- At most $d(n+2)$ prefixes are possible for a string representing a monomial.
- Hence, the algorithm invokes `Interpolate` for at most $O(td(n+2))$ times.

Running time of the algorithm

- The algorithm calls `Interpolate` on u only if u is the prefix of some string corresponding to a monomial in f.
- At most $d(n+2)$ prefixes are possible for a string representing a monomial.
- Hence, the algorithm invokes `Interpolate` for at most $O(td(n+2))$ times.

Running time of the algorithm

- The algorithm calls `Interpolate` on u only if u is the prefix of some string corresponding to a monomial in f.
- At most $d(n+2)$ prefixes are possible for a string representing a monomial.
- Hence, the algorithm invokes `Interpolate` for at most $O(td(n+2))$ times.

Interpolation of Algebraic Branching Programs

Definition (Nisan 1991, Raz-Shpilka 2005)

- An Algebraic Branching Program (ABP) is a directed acyclic graph with one vertex of in-degree zero, called the source, and a vertex of out-degree zero, called the sink.
- The vertices of the graph are partitioned into levels numbered $0, 1, \cdots, d$. Edges may only go from level i to level $i+1$ for $i \in \{0, \cdots, d-1\}$.
- The source is the only vertex at level 0 and the sink is the only vertex at level d.
- Each edge is labelled with a homogeneous linear form in the input variables. The size of the ABP is the number of vertices.

Interpolation of Algebraic Branching Programs

Definition (Nisan 1991, Raz-Shpilka 2005)

- An Algebraic Branching Program (ABP) is a directed acyclic graph with one vertex of in-degree zero, called the source, and a vertex of out-degree zero, called the sink.
- The vertices of the graph are partitioned into levels numbered $0, 1, \cdots, d$. Edges may only go from level i to level $i+1$ for $i \in \{0, \cdots, d-1\}$.
- The source is the only vertex at level 0 and the sink is the only vertex at level d.
- Each edge is labelled with a homogeneous linear form in the input variables. The size of the ABP is the number of vertices.

Interpolation of Algebraic Branching Programs

Definition (Nisan 1991, Raz-Shpilka 2005)

- An Algebraic Branching Program (ABP) is a directed acyclic graph with one vertex of in-degree zero, called the source, and a vertex of out-degree zero, called the sink.
- The vertices of the graph are partitioned into levels numbered $0, 1, \cdots, d$. Edges may only go from level i to level $i+1$ for $i \in \{0, \cdots, d-1\}$.
- The source is the only vertex at level 0 and the sink is the only vertex at level d.
- Each edge is labelled with a homogeneous linear form in the input variables. The size of the ABP is the number of vertices.

Interpolation of Algebraic Branching Programs

Definition (Nisan 1991, Raz-Shpilka 2005)

- An Algebraic Branching Program (ABP) is a directed acyclic graph with one vertex of in-degree zero, called the source, and a vertex of out-degree zero, called the sink.
- The vertices of the graph are partitioned into levels numbered $0, 1, \cdots, d$. Edges may only go from level i to level $i+1$ for $i \in \{0, \cdots, d-1\}$.
- The source is the only vertex at level 0 and the sink is the only vertex at level d.
- Each edge is labelled with a homogeneous linear form in the input variables. The size of the ABP is the number of vertices.

Algebraic Branching Program, (Nisan 1991, Raz-Shpilka 2005)

- Each of the directed paths from source to sink computes a product of linear forms. The polynomial computed by the ABP is the sum of all such product of linear forms.

Our Problem

- We are given as input an ABP P in the black-box setting.
- Our task is to output an ABP P' that computes the same polynomial as P.
- We assume that we are allowed to evaluate P at any of its intermediate gates.

Our Problem

- We are given as input an ABP P in the black-box setting.
- Our task is to output an ABP P' that computes the same polynomial as P.
- We assume that we are allowed to evaluate P at any of its intermediate gates.

Our Problem

- We are given as input an ABP P in the black-box setting.
- Our task is to output an ABP P' that computes the same polynomial as P.
- We assume that we are allowed to evaluate P at any of its intermediate gates.

The Result

- We show a polynomial time interpolation algorithm for ABPs.
- Our algorithm is motivated by Raz-Shpilka's noncommutative identity testing algorithm for formulas (and for ABP's).

The Result

- We show a polynomial time interpolation algorithm for ABPs.
- Our algorithm is motivated by Raz-Shpilka's noncommutative identity testing algorithm for formulas (and for ABP's).

Outline of the Algorithm

- Our idea is to construct the output ABP P' layer by layer such that every gate of P' computes the same polynomial as the corresponding gate in P.
- This task is trivial at level 0.
- Inductively, we assume that we have constructed P' up to layer i.

Outline of the Algorithm

- Our idea is to construct the output ABP P' layer by layer such that every gate of P' computes the same polynomial as the corresponding gate in P.
- This task is trivial at level 0.
- Inductively, we assume that we have constructed P' up to layer i.

Outline of the Algorithm

- Our idea is to construct the output ABP P' layer by layer such that every gate of P' computes the same polynomial as the corresponding gate in P.
- This task is trivial at level 0.
- Inductively, we assume that we have constructed P' up to layer i.

Outline of the Algorithm

- To interpolate P' up to layer $i+1$, we need to compute linear forms between layer i and $i+1$.
- In general we can compute the linear forms by solving exponential number of linear constraints.
- Setting up the linear constraints crucially use the fact that we can evaluate any intermediate gates of P.

Outline of the Algorithm

- To interpolate P' up to layer $i+1$, we need to compute linear forms between layer i and $i+1$.
- In general we can compute the linear forms by solving exponential number of linear constraints.
- Setting up the linear constraints crucially use the fact that we can evaluate any intermediate gates of P.

Outline of the Algorithm

- To interpolate P' up to layer $i+1$, we need to compute linear forms between layer i and $i+1$.
- In general we can compute the linear forms by solving exponential number of linear constraints.
- Setting up the linear constraints crucially use the fact that we can evaluate any intermediate gates of P.

Outline of the Algorithm

- A suitable application of Raz-Shpilka's idea provides us only a polynomial number of linear constraints that to be solved for identifying the linear forms.

Derandomizing the noncommutative identity Testing

- Bogdanov and Wee (2005) showed a randomized polynomial-time identity testing algorithm for noncommutative circuit computing small degree polynomial.
- Can one give a deterministic polynomial-time identity testing algorithm for noncommutative *circuits* computing small degree polynomial?

Derandomizing the noncommutative identity Testing

- Bogdanov and Wee (2005) showed a randomized polynomial-time identity testing algorithm for noncommutative circuit computing small degree polynomial.
- Can one give a deterministic polynomial-time identity testing algorithm for noncommutative *circuits* computing small degree polynomial?

Connection to circuit lower bound

- Analogous to the commutative case (Impagliazzo and Kabanets 2003), we observe that such an algorithm will imply either $\mathrm{NEXP} \not\subset \mathrm{P/poly}$ or the *noncommutative* Permanent function does not have polynomial-size noncommutative circuits.

Commutative *PIT* over ring

Definition

Let R be a finite commutative ring with unity and C be an arithmetic circuit in the input variable $x_1, x_2, \cdots, x_n$ over R. C computes a polynomial f in $R[x_1, x_2, \cdots, x_n]$. Suppose the operations over R can be done efficiently. Can one determine whether the polynomial computed by C is identically zero ?

Known results for *PIT* over rings

- Agrawal-Biswas (2003) showed a randomized polynomial-time algorithm for the identity testing over $\mathbb{Z}_n$.

Our Main Result

- A randomized polynomial-time identity testing algorithm over any finite commutative ring with unity where ring operations can be done efficiently.
- Conceptually and technically our result is a generalization of Agrawal-Biswas idea over arbitrary commutative ring with unity.

Outline of our algorithm

- (Univariate substitution, Agrawal-Biswas 2003) For each $x_i \leftarrow x^{(d+1)^{i-1}}$ (d be an upper bound on the degree of f).
- $g(x) \leftarrow C(x, x^{(d+1)}, \cdots, x^{(d+1)^{n-1}})$.
- $D \leftarrow d(d+1)^{n-1}$.
- Choose a monic polynomial $q(x)$ (whose coefficients are multiple of unity) of degree $\lceil \log\ 24D \rceil$ uniformly at random.

Outline of our algorithm

- (Univariate substitution, Agrawal-Biswas 2003) For each $x_i \leftarrow x^{(d+1)^{i-1}}$ (d be an upper bound on the degree of f).
- $g(x) \leftarrow C(x, x^{(d+1)}, \cdots, x^{(d+1)^{n-1}})$.
- $D \leftarrow d(d+1)^{n-1}$.
- Choose a monic polynomial $q(x)$ (whose coefficients are multiple of unity) of degree $\lceil \log 24D \rceil$ uniformly at random.

Outline of our algorithm

- (Univariate substitution, Agrawal-Biswas 2003) For each $x_i \leftarrow x^{(d+1)^{i-1}}$ (d be an upper bound on the degree of f).
- $g(x) \leftarrow C(x, x^{(d+1)}, \cdots, x^{(d+1)^{n-1}})$.
- $D \leftarrow d(d+1)^{n-1}$.
- Choose a monic polynomial $q(x)$ (whose coefficients are multiple of unity) of degree $\lceil \log\ 24D \rceil$ uniformly at random.

Outline of our algorithm

- (Univariate substitution, Agrawal-Biswas 2003) For each $x_i \leftarrow x^{(d+1)^{i-1}}$ (d be an upper bound on the degree of f).
- $g(x) \leftarrow C(x, x^{(d+1)}, \cdots, x^{(d+1)^{n-1}})$.
- $D \leftarrow d(d+1)^{n-1}$.
- Choose a monic polynomial $q(x)$ (whose coefficients are multiple of unity) of degree $\lceil \log \ 24D \rceil$ uniformly at random.

Outline of our algorithm

- Divide $g(x)$ by $q(x)$ and compute the remainder $r(x)$.
- If $r(x) = 0$, C computes a zero polynomial.
- Else C computes a nonzero polynomial.

Outline of our algorithm

- Divide $g(x)$ by $q(x)$ and compute the remainder $r(x)$.
- If $r(x) = 0$, C computes a zero polynomial.
- Else C computes a nonzero polynomial.

Outline of our algorithm

- Divide $g(x)$ by $q(x)$ and compute the remainder $r(x)$.
- If $r(x) = 0$, C computes a zero polynomial.
- Else C computes a nonzero polynomial.

Thank You