Tutorial on Enumeration Complexity: Defining Tractability

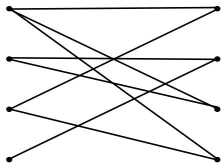
Yann Strozecki with Florent Capelli and Arnaud Mary

Dagstuhl Seminar on Enumeration in Data Management

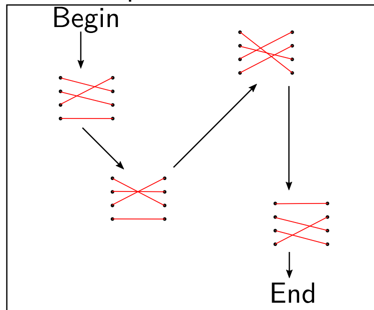
Enumeration problems

- **Enumeration problems**: list all solutions rather than deciding whether there is one or finding one.
- Complexity measures: total time and delay between solutions.
- Motivations: database queries, counting, optimization, building libraries, datamining.

Perfect matching ?



Solution space:



Framework

An enumeration problem A is a function which associates to each input a set of solutions $A(x)$.

An enumeration algorithm must generate every element of $A(x)$ one after the other without repetition.

The computation model for enumeration is a RAM with uniform cost measure and an OUPTPUT instruction. Support efficient data structures.

Complexity measures:

- total time
- delay
- space

Parameters:

- input size
- output size
- single solution size

Complexity classes

Several classes introduced in the 80's [Johnson, Yannakakis and Papadimitriou]. Preprocessing for some classes.

1. Polynomially balanced predicate: $\textsc{EnumP}$
2. Output polynomial: $\textsc{OutputP}$
3. Incremental polynomial time: $\textsc{IncP}$
4. Polynomial delay: $\textsc{DelayP}$
5. Strong polynomial delay: $\textsc{SDelayP}$
6. Constant Delay: $\textsc{CD}$

Complexity classes

Several classes introduced in the 80's [Johnson, Yannakakis and Papadimitriou]. Preprocessing for some classes.

1. Polynomially balanced predicate: $\textsc{EnumP}$
2. Output polynomial: $\textsc{OutputP}$
3. Incremental polynomial time: $\textsc{IncP}$
4. Polynomial delay: $\textsc{DelayP}$
5. Strong polynomial delay: $\textsc{SDelayP}$
6. Constant Delay: $\textsc{CD}$

Classes above $\textsc{EnumP}$ in the next talk [Kröll].

Poly time testing

Definition

A problem A is in $\textsc{EnumP}$ if deciding whether $y \in A(x)$ is in P and if all $y \in A(x)$ are of polynomial size in x.

Equivalent to the class NP.

Poly time testing

Definition

A problem A is in $\textsc{EnumP}$ if deciding whether $y \in A(x)$ is in P and if all $y \in A(x)$ are of polynomial size in x.

Equivalent to the class NP.

Definition

A **parsimonious reduction** from A to B, two enumeration problems, is a pair of polynomial time computable functions f, g such that for all x, $g(x)$ is a bijection from $B(f(x))$ to $A(x)$.

- Useful to prove hardness of enumerating solutions of NP-complete problems.
- Not general enough to prove hardness of natural problems.

Output polynomial

An output sensitive algorithm has its complexity depending on both its input and output.

Definition

A problem $A \in \textsc{EnumP}$ is in $\textsc{OutputP}$ if there is a polynomial p and a machine M which solves A in total time $O(p(|x|, |A(x)|))$.

Output polynomial

An output sensitive algorithm has its complexity depending on both its input and output.

Definition

A problem $A \in \textsc{EnumP}$ is in $\textsc{OutputP}$ if there is a polynomial p and a machine M which solves A in total time $O(p(|x|, |A(x)|))$.

$\textsc{OutputP} \neq \textsc{EnumP}$ iff P $\neq$ NP, using enumeration of solutions of any NP-complete problem.

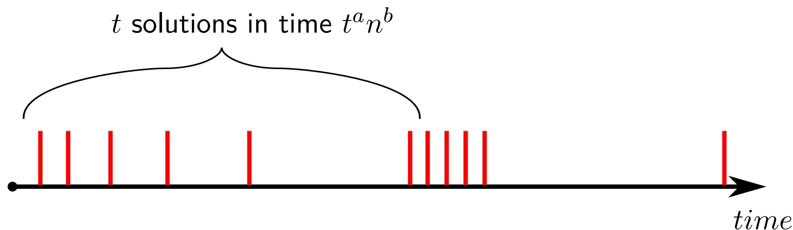
Question: is there a natural problem in $\textsc{OutputP}$ but not in the classes below?

Incremental time

A machine M enumerates A in *incremental time* $f(t)g(n)$ if on every input x, M enumerates t elements of $A(x)$ in time $f(t)g(|x|)$ for every $t \leq |A(x)|$.

Definition (Incremental polynomial time)

IncP is the set of enumeration problems such that there is an algorithm in incremental time $O(t^a n^b)$, for inputs of size n and a, b constants.



Saturation algorithm

Most algorithms in incremental time are saturation algorithms:

- **begin** with a polynomial number of simple solutions
- **for each** k-uple of already generated solutions apply a rule to produce a new solution
- **stop** when no new solutions are found

Saturation algorithm

Most algorithms in incremental time are saturation algorithms:

- **begin** with a polynomial number of simple solutions
- **for each** k-uple of already generated solutions apply a rule to produce a new solution
- **stop** when no new solutions are found

1. Accessible vertices in a graph by flooding.
2. Generate a finite group from a set of generators.
3. Generating all the circuits of a matroid.
4. Generate all possible unions of sets:

    - $\{12, 134, 23, 14\}$
    - $\{12, 134, 1234, 23, 14\}$
    - $\{12, 134, 1234, 23, 123, 14\}$
    - $\{12, 134, 1234, 23, 123, 14, 124\}$

Relation to search problem

Search problem $\textsc{AnotherSol}{\cdot}A$
Input: x and a set of solutions $S \subset A(x)$
Output: $y \in A(x) \setminus S$ or $\sharp$ if there is none.

Relation to search problem

Search problem $\textsc{AnotherSol}{\cdot}A$
Input: x and a set of solutions $S \subset A(x)$
Output: $y \in A(x) \setminus S$ or $\sharp$ if there is none.

Theorem

An enumeration problem A is in $\textsc{IncP}$ if and only if $\textsc{AnotherSol}{\cdot}A \in$ FP.

Used for hardness proof: the generation of maximal models of Horn formulas [Kavvadias et al.], dualization in distributive lattice [Babin and Kuznetsov, Defrain and Nourine].

Relationship with total functions

Definition

A problem in TFNP is a polynomially balanced polynomial time predicate A such that for all x, $A(x)$ is not empty. An algorithm solving A must produce an element of $A(x)$ on input x.

$$\mathsf{TFNP} = \mathsf{FP}^{\mathsf{NP} \cap \mathsf{coNP}}$$

Relationship with total functions

Definition

A problem in TFNP is a polynomially balanced polynomial time predicate A such that for all x, $A(x)$ is not empty. An algorithm solving A must produce an element of $A(x)$ on input x.

$$\mathsf{TFNP} = \mathsf{FP}^{\mathsf{NP} \cap \mathsf{coNP}}$$

Proposition (Capelli, S. 2018)

TFNP = FP *if and only if* $\textsc{IncP} = \textsc{OutputP}$.

Proof: ($\Rightarrow$)Remark that $\textsc{AnotherSol}{\cdot}A$ is a TFNP problem when $A \in \textsc{OutputP}$.
($\Leftarrow$) Use many distinct copies of $A(x)$ to obtain an $\textsc{OutputP}$ problem, an $\textsc{IncP}$ algorithm allows to find one solution in FP.

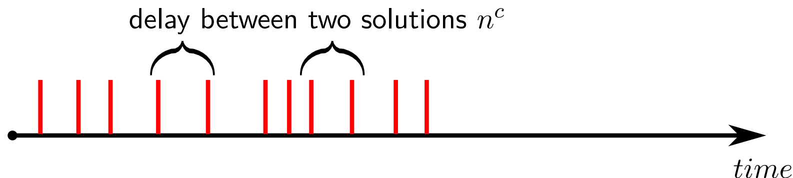
Polynomial Delay

The delay is the maximum time between the production of two consecutive solutions in an enumeration.

Definition (Polynomial delay)

DelayP is the set of enumeration problems solved by an algorithm whose delay is polynomial in the input.

$$\textsc{DelayP} \subseteq \textsc{IncP}$$



Reduction for DelayP problems

Proposition (Durand, S.)

Let A and B be two problems in $\textsc{DelayP}$ *then* $A \cup B$ *is in* $\textsc{DelayP}$.

Reduction for DelayP problems

Proposition (Durand, S.)

Let A and B be two problems in $\textsc{DelayP}$ then $A \cup B$ is in $\textsc{DelayP}$.

Proof sketch: Output a solution of A if it is not a solution of B otherwise output a solution of B.

If the solutions are generated in the same order, just merge them dynamically.

Reduction for DelayP problems

Proposition (Durand, S.)

Let A and B be two problems in $\textsc{DelayP}$ *then* $A \cup B$ *is in* $\textsc{DelayP}$*.*

Proof sketch: Output a solution of A if it is not a solution of B otherwise output a solution of B.

If the solutions are generated in the same order, just merge them dynamically.

Definition (Polynomial delay reduction)

Turing reduction with only cartesian products and unions keeps $\textsc{DelayP}$ stable.

Similar to d-DNNF set circuits [Amarilli et al.].

Relationship between incremental and polynomial delay

Definition (Incremental polynomial time hierarchy)

A problem $A \in \textsc{EnumP}$ is in $\textsc{IncP}_a$ if there is a machine M which solves it in incremental time $O(t^a n^b)$ for some constant b.

Relationship between incremental and polynomial delay

Definition (Incremental polynomial time hierarchy)

A problem $A \in \textsc{EnumP}$ is in $\textsc{IncP}_a$ if there is a machine M which solves it in incremental time $O(t^a n^b)$ for some constant b.

Proposition

$\textsc{IncP}_1 = \textsc{DelayP}$.

Amortize generation of solutions, using a large queue: exponential memory.

Are IncP_1 and DelayP really equal?

The main difference between $\textsc{IncP}_1$ and $\textsc{DelayP}$ is the regularity of the delays or equivalently the memory usage.

Are IncP$_1$ and DelayP really equal?

The main difference between $\textsc{IncP}_1$ and $\textsc{DelayP}$ is the regularity of the delays or equivalently the memory usage.

Theorem (Capelli, S. 2018)

Let A be a problem with a polynomial space incremental linear algorithm such that $\forall t < |A(x)|$, a polynomial fraction of the first t solutions are generated with polynomial delay. Then $A \in \textsc{DelayP}$ with polynomial space.

Proof sketch: Simulate the algorithm at different points in time and use the parts with high density of solutions to compensate for sparse parts.

Are IncP_1 and DelayP really equal?

The main difference between $\textsc{IncP}_1$ and $\textsc{DelayP}$ is the regularity of the delays or equivalently the memory usage.

Theorem (Capelli, S. 2018)

Let A be a problem with a polynomial space incremental linear algorithm such that $\forall t < |A(x)|$, a polynomial fraction of the first t solutions are generated with polynomial delay. Then $A \in \textsc{DelayP}$ with polynomial space.

Proof sketch: Simulate the algorithm at different points in time and use the parts with high density of solutions to compensate for sparse parts.

Open problem: Generalize Cheater's lemma [Carmeli]. Turn a polynomial delay and polynomial space enumeration algorithm with polynomial repetitions into a proper poly delay and poly space algorithm.

Separation of DelayP and IncP

Corollary (Capelli, S. 2018)

If ETH holds, then $\textsc{DelayP} \subsetneq \textsc{IncP}$.

Separation of DelayP and IncP

Corollary (Capelli, S. 2018)

If ETH holds, then $\textsc{IncP}_1 \subsetneq \textsc{IncP}_a$ for $a > 1$.

Separation of DelayP and IncP

Corollary (Capelli, S. 2018)

If ETH holds, then $\textsc{IncP}_1 \subsetneq \textsc{IncP}_a$ for $a > 1$.

Theorem (Capelli, S. 2018)

If ETH holds, then $\textsc{IncP}_a \subsetneq \textsc{IncP}_b$ for all $a < b$.

Proof sketch: Problem Pad_t, input φ a CNF, with 2^{nt} trivial solutions and the models of φ duplicated 2^n times.
Since $\textsc{IncP}_a = \textsc{IncP}_b$, Pad_{b-1} gives a $O(2^{\frac{a}{b}n})$ algorithm to solve SAT.
Using the better SAT algorithm, we have $Pad_{\frac{a}{b^2}} \in \textsc{IncP}_b$. Repeat this trick to contradict ETH.

Separation of DelayP and IncP

Corollary (Capelli, S. 2018)

If ETH holds, then $\textsc{IncP}_1 \subsetneq \textsc{IncP}_a$ for $a > 1$.

Theorem (Capelli, S. 2018)

If ETH holds, then $\textsc{IncP}_a \subsetneq \textsc{IncP}_b$ for all $a < b$.

Proof sketch: Problem Pad_t, input φ a CNF, with 2^{nt} trivial solutions and the models of φ duplicated 2^n times.
Since $\textsc{IncP}_a = \textsc{IncP}_b$, Pad_{b-1} gives a $O(2^{\frac{a}{b}n})$ algorithm to solve SAT.
Using the better SAT algorithm, we have $Pad_{\frac{a}{b^2}} \in \textsc{IncP}_b$. Repeat this trick to contradict ETH.

Open question: is there a natural problem in $\textsc{IncP}$ not in $\textsc{DelayP}$? Do you have any candidate problem?

Restricting IncP: when is saturation in polynomial delay

Question

Can we solve saturation problems in polynomial delay ?

Restricting IncP: when is saturation in polynomial delay

Question

Can we solve saturation problems in polynomial delay ?

Yes sometimes, for saturation of sets by union, using binary partition it can be done in linear delay.

No in general, since saturation problems are "equal" to $\textsc{IncP}$ and $\textsc{IncP} \neq \textsc{DelayP}$.

Restricting IncP: when is saturation in polynomial delay

Question

Can we solve saturation problems in polynomial delay ?

Yes sometimes, for saturation of sets by union, using binary partition it can be done in linear delay.

No in general, since saturation problems are "equal" to IncP and IncP $\neq$ DelayP.

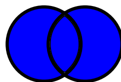
"CSP style": restrict the saturation rules, consider set operations.

Set operations

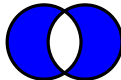
A set over $\{1, \ldots, n\}$ is represented by its characteristic vector.

A set operation is a boolean operation $\{0,1\}^k \to \{0,1\}$ applied componentwise to k boolean vectors.

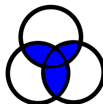
$\vee$ $\quad \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \vee \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$ $\quad \cup$



$+$ $\quad \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$ $\quad \triangle$



$\mathrm{maj}(x, y, z)$ $\quad \mathrm{maj}(\begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}) = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$ Majority



Closure by set operation

Let S be a set of boolean vectors of size n and $\mathcal{F}$ be a finite set of boolean operations.

Closure:

- $\mathcal{F}^0(\mathsf{S}) = \mathsf{S}$
- $\mathcal{F}^i(\mathsf{S}) = \mathcal{F}^{i-1}(\mathsf{S}) \cup$ $\{f(v_1, \ldots, v_t) \mid v_1, \ldots, v_t \in \mathcal{F}^{i-1}(S)$ and $f \in \mathcal{F}\}$
- $Cl_{\mathcal{F}}(\mathsf{S}) = \cup_i \mathcal{F}^i(\mathsf{S})$

The enumeration problem is to list the elements of $Cl_{\mathcal{F}}(\mathsf{S})$.

Classification

We classify all sets of set operations using Post lattice. It reduces the problem to a few cases:

- addition (vector space)
- disjunction, conjunction (boolean algebra)
- near unanimity terms (caracterized by projection)
- disjunction (hardest complexity-wise)

Classification

We classify all sets of set operations using Post lattice. It reduces the problem to a few cases:

- addition (vector space)
- disjunction, conjunction (boolean algebra)
- near unanimity terms (caracterized by projection)
- disjunction (hardest complexity-wise)

Theorem (Mary, S. 2016)

Let $\mathcal{F}$ be a set of set operations, then listing the elements of $Cl_{\mathcal{F}}(\mathsf{S})$ can be done with delay polynomial in S.

Work either by the flashlight search, reduction to a simple SAT formulas or Gray code enumeration for simple algebraic structure.

Capturing more saturation operations

1. Larger domain.
2. Non uniform operation, acting differently on each element.
3. Enumerating maximal/minimal elements only.

Capturing more saturation operations

1. Larger domain.
2. Non uniform operation, acting differently on each element.
3. Enumerating maximal/minimal elements only.

(1) Uncountably many new cases. NP-hard extension problem but polynomial time for several algebraic operators [Bulatov et al.]. For binary associative operators possible with exponential space.

Capturing more saturation operations

1. Larger domain.
2. Non uniform operation, acting differently on each element.
3. Enumerating maximal/minimal elements only.

(1) Uncountably many new cases. NP-hard extension problem but polynomial time for several algebraic operators [Bulatov et al.]. For binary associative operators possible with exponential space.

(2) If all operators act on a single coefficient (downward/upward closure), everything is polynomial delay. For operators acting on 3 coefficients, harder than $\textsc{EnumP}$.

Capturing more saturation operations

1. Larger domain.
2. Non uniform operation, acting differently on each element.
3. Enumerating maximal/minimal elements only.

(1) Uncountably many new cases. NP-hard extension problem but polynomial time for several algebraic operators [Bulatov et al.]. For binary associative operators possible with exponential space.

(2) If all operators act on a single coefficient (downward/upward closure), everything is polynomial delay. For operators acting on 3 coefficients, harder than $\textsc{EnumP}$.

(3) Equivalent to fundamental problems in $\textsc{IncP}$[Mary, S. 2019].

What is a really efficient enumeration algorithm ?

- DelayP: equivalent to P for enumeration.
- SDelayP: polynomial delay in the size of the last solution.
- CD: constant delay.

What is a really efficient enumeration algorithm ?

- DelayP: equivalent to P for enumeration.
- SDelayP: polynomial delay in the size of the last solution.
- CD: constant delay.
- Additional space polynomial in the input or a solution.
- Polynomial time sampling.

What is a really efficient enumeration algorithm ?

- DelayP: equivalent to P for enumeration.
- SDelayP: polynomial delay in the size of the last solution.
- CD: constant delay.
- Additional space polynomial in the input or a solution.
- Polynomial time sampling.

Help through relaxations:

- Randomized algorithms.
- Average delay: Total time / Number of solutions.
- Approximate enumeration.

Four flavors of constant delay

The term constant delay is used to denote different things.

- **Real** constant delay (Gray code like algorithms).
  Enumeration goes from a solution to the next while changing a constant number of bits.

Four flavors of constant delay

The term constant delay is used to denote different things.

- **Real** constant delay (Gray code like algorithms).
  Enumeration goes from a solution to the next while changing a constant number of bits.
- Constant amortized time (CAT) algorithms. Generation of combinatorial structures of a given size, subgraphs of graphs. Pushout amortization [Uno].

Four flavors of constant delay

The term constant delay is used to denote different things.

- **Real** constant delay (Gray code like algorithms). Enumeration goes from a solution to the next while changing a constant number of bits.
- Constant amortized time (CAT) algorithms. Generation of combinatorial structures of a given size, subgraphs of graphs. Pushout amortization [Uno].
- Allow dynamic amortization (generalized OUPTUT instruction).

Four flavors of constant delay

The term constant delay is used to denote different things.

- **Real** constant delay (Gray code like algorithms). Enumeration goes from a solution to the next while changing a constant number of bits.
- Constant amortized time (CAT) algorithms. Generation of combinatorial structures of a given size, subgraphs of graphs. Pushout amortization [Uno].
- Allow dynamic amortization (generalized OUPTUT instruction).
- FPT algorithm, arbitrary dependency in the parameter. Many examples from logic/database (data complexity) [Segoufin]. Often polynomial number of solutions: restricting preprocessing is fundamental.

The case for strong polynomial delay

Strong polynomial delay is important when the input is large with regard to the size of one solution.

Relevant for problems on hypergraphs or for infinite enumeration where the size of the solutions grows arbitrarily.

The case for strong polynomial delay

Strong polynomial delay is important when the input is large with regard to the size of one solution.

Relevant for problems on hypergraphs or for infinite enumeration where the size of the solutions grows arbitrarily.

Why is it so rarely considered ?

1. People are satisfied/used to polynomial delay.
2. Harder to obtain often because of repetitions.
3. In graph problems, typically the instance is of size $m = O(n^2)$ and the solutions are of size n: not a complexity problem.

The case for strong polynomial delay

Strong polynomial delay is important when the input is large with regard to the size of one solution.

Relevant for problems on hypergraphs or for infinite enumeration where the size of the solutions grows arbitrarily.

Why is it so rarely considered ?

1. People are satisfied/used to polynomial delay.
2. Harder to obtain often because of repetitions.
3. In graph problems, typically the instance is of size $m = O(n^2)$ and the solutions are of size n: not a complexity problem.

Much easier to prove lower bound for strong polynomial delay.

Enumerating the models of a DNF

- A term is a conjunction of literals over n variables.
- A DNF formula is a disjunction of m terms.
- $\textsc{Enum}{\cdot}DNF$ is the problem of enumerating satisfying assignments of a DNF.

Why is this problem interesting?

Enumerating the models of a DNF

- A term is a conjunction of literals over n variables.
- A DNF formula is a disjunction of m terms.
- $\textsc{Enum}{\cdot}DNF$ is the problem of enumerating satisfying assignments of a DNF.

Why is this problem interesting?

- Extremely simple: solution of terms in constant delay. Union of regular sets of solutions while dealing with repetitions.
- DNF enumeration is connected to knowledge representation, minimal transversal enumeration, subset membership queries, CQ + SO variables, DNF model counting, PAC-learning . . .

Lower Bound Conjectures

Best complexity by binary partition (similar to monotone CNF [Uno]) delay linear in $O(mn)$. Can we get rid of m ?

Lower Bound Conjectures

Best complexity by binary partition (similar to monotone CNF [Uno]) delay linear in $O(mn)$. Can we get rid of m ?

DNF Enumeration Conjecture

Generating the models of a DNF is not in $\textsc{SDelayP}$.

Strong DNF Enumeration Conjecture

There is no algorithm generating the models of a DNF in delay $o(m)$ where m is the number of terms.

Lower Bound Conjectures

Best complexity by binary partition (similar to monotone CNF [Uno]) delay linear in $O(mn)$. Can we get rid of m ?

DNF Enumeration Conjecture

Generating the models of a DNF is not in $\textsc{SDelayP}$.

Strong DNF Enumeration Conjecture

There is no algorithm generating the models of a DNF in delay $o(m)$ where m is the number of terms.

The conjectures can be made stronger by looking at subclasses of DNF and average delay.

Results [Capelli, S. 2019]

Class	Delay	Space
DNF	$O(\|\|D\|\|)$	$O(\|\|D\|\|)$
$(\star)$ DNF	$O(n\sqrt{m})$ average delay	$O(\|\|D\|\|)$
$(\star)$ k-DNF	$2^{O(k)}$	$O(\|\|D\|\|)$
$(\star)$ Monotone DNF	$O(n^2)$, m^2 preprocessing	$O(\|Output\|)$
$(\star)$ Monotone DNF	$O(\log(n)\log(nm))$ average delay	$O(mn)$

Table: Overview of the results. In this table, D is a DNF, n its number of variables and m its number of terms. New contributions are annotated with $(\star)$.

Summary

$$\mathrm{CD} \subseteq \textsc{SDelayP} \subseteq \textsc{DelayP} \subsetneq \textsc{IncP} \subsetneq \textsc{OutputP} \subsetneq \textsc{EnumP}$$

Conditional separation under complexity hypotheses: $\mathsf{P} \neq \mathsf{NP}$, $\mathsf{TFNP} \neq \mathsf{FP}$ and ETH.

Summary

$$\mathrm{CD} \subsetneq \mathrm{SD{\scriptsize ELAY}P} \subsetneq \mathrm{D{\scriptsize ELAY}P} \subsetneq \mathrm{I{\scriptsize NC}P} \subsetneq \mathrm{O{\scriptsize UTPUT}P}$$

If we remove the condition to be in $\mathrm{E{\scriptsize NUM}P}$: unconditional separation.
Benny: example of frequent isomorphic subgraphs -> easy to enumerate, hard to generate (find other exemples)

Open problems

Lower bounds for real problems using (S)ETH for $\textsc{SDelayP}$, $\textsc{DelayP}$, $\textsc{IncP}_a$:

1. Minimal hitting sets of hypergraphs: delay of $m^{O(\log(m))}$.
2. Minimal hitting sets of k-regular hypergraphs in $\textsc{IncP}_{k+2}$.
3. Maximal cliques of a graph in delay $O(mn)$.
4. Circuits of a binary matroids in $\textsc{IncP}_2$.

Open problems

Lower bounds for real problems using (S)ETH for $\textsc{SDelayP}$, $\textsc{DelayP}$, $\textsc{IncP}_a$:

1. Minimal hitting sets of hypergraphs: delay of $m^{O(\log(m))}$.
2. Minimal hitting sets of k-regular hypergraphs in $\textsc{IncP}_{k+2}$.
3. Maximal cliques of a graph in delay $O(mn)$.
4. Circuits of a binary matroids in $\textsc{IncP}_2$.

Discussion topics for enumeration:

1. Revisit classical problems focusing on amortized delay or linear incremental time.
2. Circuits for composing constant/polynomial delay algorithms: better algorithms and reductions. Efficient solution representation.
3. Diverse solution enumeration: generate a good covering of the solution space.

Thanks !

Questions ?

Link between uniform generators and enumeration

A uniform generator for the problem A is an algorithm, which given x samples the elements of $A(x)$ with uniform probability.

Theorem

If $A \in \textsc{EnumP}$ has a polytime uniform generator, then A is in randomized $\textsc{DelayP}$.

The space is proportional to the number of solutions but can be improved if we accept repetitions.

Proposition

If $A \in \textsc{EnumP}$ has a polytime uniform generator, then there is an enumeration algorithm in randomized $\textsc{IncP}_1$ with repetitions and polynomial space.