Efficient Enumeration Algorithms via Circuits

Antoine Amarilli
October 18, 2023

Télécom Paris
General motivation (for database people)

• **Intensional query evaluation**: given a query $Q$ and a database $D$

Monday: intensional query evaluation for counting and probability computation

Today: can we use the intensional approach to enumerate query answers?

Structure of the talk:

• Preliminaries and problem statement
• Efficient enumeration for $d$-DNNF set circuits
• Applications: Using enumeration on circuits for query evaluation
• **Intensional query evaluation**: given a query $Q$ and a database $D$
  • Compile $Q$ on $D$ to a circuit $C$ in a tractable class (d-SDNNF, uOBDD, ...)
    • Other names: *grounding* $Q$ on $D$, computing the *provenance* of $Q$ on $D$...
General motivation (for database people)

- **Intensional query evaluation**: given a query $Q$ and a database $D$
  - Compile $Q$ on $D$ to a circuit $C$ in a tractable class (d-SDNNF, uOBDD, ...)
    - Other names: *grounding* $Q$ on $D$, computing the *provenance* of $Q$ on $D$...
  - **Use** the circuit $C$ to retrieve the answer of $Q$ on $D$
General motivation (for database people)

- **Intensional query evaluation**: given a query $Q$ and a database $D$
  - Compile $Q$ on $D$ to a circuit $C$ in a tractable class (d-SDNNF, uOBDD, ...)
  - Other names: grounding $Q$ on $D$, computing the provenance of $Q$ on $D$...
  - Use the circuit $C$ to retrieve the answer of $Q$ on $D$

- **Monday**: intensional query evaluation for **counting** and **probability computation**
General motivation (for database people)

• **Intensional query evaluation**: given a query $Q$ and a database $D$
  • Compile $Q$ on $D$ to a circuit $C$ in a tractable class (d-SDNNF, uOBDD, ...)
    • Other names: grounding $Q$ on $D$, computing the provenance of $Q$ on $D$...
  • Use the circuit $C$ to retrieve the answer of $Q$ on $D$

• **Monday**: intensional query evaluation for counting and probability computation

• **Today**: can we use the intensional approach to enumerate query answers?
General motivation (for database people)

- **Intensional query evaluation**: given a query $Q$ and a database $D$
  - Compile $Q$ on $D$ to a circuit $C$ in a tractable class (d-SDNNF, uOBDD, ...)
    - Other names: grounding $Q$ on $D$, computing the provenance of $Q$ on $D$...
  - Use the circuit $C$ to retrieve the answer of $Q$ on $D$

- **Monday**: intensional query evaluation for counting and probability computation

- **Today**: can we use the intensional approach to enumerate query answers?

Structure of the talk:

- Preliminaries and problem statement
General motivation (for database people)

- **Intensional query evaluation**: given a query $Q$ and a database $D$
  - Compile $Q$ on $D$ to a circuit $C$ in a tractable class (d-SDNNF, uOBDD, ...)
    - Other names: grounding $Q$ on $D$, computing the provenance of $Q$ on $D$...
  - Use the circuit $C$ to retrieve the answer of $Q$ on $D$

- **Monday**: intensional query evaluation for counting and probability computation
- **Today**: can we use the intensional approach to enumerate query answers?

Structure of the talk:

- Preliminaries and problem statement
- Efficient enumeration for d-DNNF set circuits
General motivation (for database people)

- **Intensional query evaluation**: given a query $Q$ and a database $D$
  - Compile $Q$ on $D$ to a circuit $C$ in a tractable class (d-SDNNF, uOBDD, ...)
  - Other names: **grounding** $Q$ on $D$, computing the **provenance** of $Q$ on $D$...
  - **Use** the circuit $C$ to retrieve the answer of $Q$ on $D$

- **Monday**: intensional query evaluation for *counting* and *probability computation*

- **Today**: can we use the intensional approach to *enumerate* query answers?

Structure of the talk:

- Preliminaries and problem statement
- Efficient enumeration for **d-DNNF set circuits**
- Applications: Using enumeration on circuits for **query evaluation**
Dramatis Personae

Antoine Amarilli
Pierre Bourhis
Florent Capelli
Louis Jachiet
Stefan Mengel

Mikaël Monet
Martín Muñoz
Matthias Niewerth
Cristian Riveros
Amarilli, A., Bourhis, P., Jachiet, L., and Mengel, S.
**A Circuit-Based Approach to Efficient Enumeration.** ICALP 2017.

Amarilli, A., Bourhis, P., and Mengel, S.
**Enumeration on Trees under Relabelings.** ICDT 2018.

Amarilli, A., Bourhis, P., Mengel, S., and Niewerth, M.
**Constant-Delay Enumeration for Nondeterministic Document Spanners.** ICDT 2019.

Amarilli, A., Bourhis, P., Mengel, S., and Niewerth, M.
**Enumeration on Trees with Tractable Combined Complexity and Efficient Updates.** PODS 2019.

Amarilli, A., Bourhis, P., Mengel, S., and Niewerth, M.

Amarilli, A., Jachiet, L., Muñoz, M., and Riveros, C.
**Efficient Enumeration for Annotated Grammars.** PODS 2022.

Amarilli, A., Bourhis, P., Capelli, F., Monet, M.
**Ranked Enumeration for MSO on Trees via Knowledge Compilation.** Under review.
Preliminaries
Enumeration algorithms (see Nofar’s talks, September workshop)

Step 1: Indexing in $O(|\text{input}|)$

Indexed input

Step 2: Enumeration in $O(|\text{result}|)$

Results

State

5/24
Enumeration algorithms (see Nofar’s talks, September workshop)

Step 1: Indexing in $O(|input|)$

Results

State $\frac{5}{24}$
Enumeration algorithms

(see Nofar’s talks, September workshop)

Step 1:
Indexing
in \(O(|\text{input}|)\)

Indexed input

Input

A B C

a
b

a'
b

b'
c

a'
b'
c

Results

State: 5/24
Enumeration algorithms (see Nofar’s talks, September workshop)

Step 1: Indexing in $O(|\text{input}|)$

Step 2: Enumeration in $O(|\text{result}|)$
Enumeration algorithms (see Nofar’s talks, September workshop)

Step 1: Indexing in $O(|\text{input}|)$

Indexed input

Step 2: Enumeration in $O(|\text{result}|)$

Results

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>
Enumeration algorithms (see Nofar’s talks, September workshop)

Step 1: Indexing in $O(|\text{input}|)$

Indexed input

Step 2: Enumeration in $O(|\text{result}|)$

Results

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td></td>
</tr>
</tbody>
</table>

State

0011
Enumeration algorithms (see Nofar’s talks, September workshop)

Step 1: Indexing in $O(|\text{input}|)$

Step 2: Enumeration in $O(|\text{result}|)$

Input

Indexed input

A B C

a b c

0011

State

Results
Enumeration algorithms (see Nofar’s talks, September workshop)

Input

Step 1:
Indexing in \(O(|\text{input}|)\)

Indexed input

Step 2:
Enumeration in \(O(|\text{result}|)\)

A B C

a' b c

Results

State

010001
Enumeration algorithms (see Nofar's talks, September workshop)

Input

Step 1: Indexing in $O(|\text{input}|)$

Indexed input

Step 2: Enumeration in $O(|\text{result}|)$

Results

State

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b'</td>
<td>c</td>
</tr>
</tbody>
</table>

01100111
Enumeration algorithms
(see Nofar's talks, September workshop)

Input

Step 1: Indexing in $O(|\text{input}|)$

Indexed input

Step 2: Enumeration in $O(|\text{result}|)$

Results

State
WITH knowledge compilation:

\[
\begin{array}{ccc}
\cup & \times & \top \\
\times & \cup & \top \\
\top & \cup & \top \\
\end{array}
\]
WITHOUT knowledge compilation:

Input → Enumeration → Results

Input → Enumeration → Results

WITH knowledge compilation:

Input → Compilation → Circuit → Enumeration → Results

\[
\begin{align*}
\text{Input} & \quad \text{Compilation} \quad \cup \quad \times \quad x \quad \top \quad \times \quad z \\
\text{Circuit} & \quad \cup \quad \times \quad x \quad \top \quad \times \quad z \\
\text{Input} & \quad \text{Compilation} \quad \cup \quad \times \quad x \quad \top \quad \times \quad z
\end{align*}
\]
Knowledge compilation
(see Guy and YooJung’s talks, Boot camp)

WITHOUT knowledge compilation:

Input → Enumeration → Results

Input → Enumeration → Results

Input → Enumeration → Results

WITH knowledge compilation:

Input → Compilation → Circuit → Enumeration → Results

\[
\begin{array}{ccc}
A & B & C \\
a & b & c \\
a & b' & c \\
\end{array}
\]

\[
\begin{array}{ccc}
A & B & C \\
a & b & c \\
a & b' & c \\
\end{array}
\]

\[
\begin{array}{ccc}
A & B & C \\
a & b & c \\
a & b' & c \\
\end{array}
\]

\[
\begin{array}{ccc}
A & B & C \\
a & b & c \\
a & b' & c \\
\end{array}
\]
Knowledge compilation

WITHOUT knowledge compilation:

Input

Enumeration

\[
\begin{array}{ccc}
A & B & C \\
\hline
a & b & c \\
a & b' & c \\
\end{array}
\]

Results

WITH knowledge compilation:

Input

Compilation

\[
\begin{array}{ccc}
\cup & \times & x \\
\top & \times & z \\
\end{array}
\]

Circuit

\[
\begin{array}{ccc}
A & B & C \\
\hline
a & b & c \\
a & b' & c \\
\end{array}
\]

Results
Knowledge compilation (see Guy and YooJung’s talks, Boot camp)

**WITHOUT knowledge compilation:**

```
Input Enumeration

A B C
a b c a' b' c

Results
```

**WITH knowledge compilation:**

```
Input Compilation Circuit

\[ a \cup b \times x \top \times z \]

Input Compilation Circuit

\[ a \cup b \times x \top \times z \]

Results
```

\[ \frac{6}{24} \]
Knowledge compilation

WITHOUT knowledge compilation:

Input ➔ Enumeration ➔ Results

Input ➔ Enumeration ➔ Results

Input ➔ Enumeration ➔ Results

WITH knowledge compilation:

Input ➔ Compilation ➔ Circuit

Input ➔ Compilation ➔ Circuit

Input ➔ Compilation ➔ Circuit

(see Guy and YooJung’s talks, Boot camp)
WITHOUT knowledge compilation:

<table>
<thead>
<tr>
<th>Input</th>
<th>Enumeration</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>a b c</td>
<td>a b' c</td>
</tr>
</tbody>
</table>

WITH knowledge compilation:

<table>
<thead>
<tr>
<th>Input</th>
<th>Compilation</th>
<th>Circuit</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>a b c</td>
<td>a b' c</td>
</tr>
</tbody>
</table>

(see Guy and YooJung’s talks, Boot camp)
Directed acyclic graph of gates
Set circuits

- Directed acyclic graph of gates
- Output gate: 

Factorized database fans may find these eerily familiar
Set circuits

- Directed acyclic graph of **gates**
- **Output** gate:
- **Variable** gates:

Factorized database fans may find these eerily familiar.
Set circuits

- Directed acyclic graph of gates
- Output gate:
- Variable gates:
- Constant gates:

Internal gates may find these eerily familiar.
- Directed acyclic graph of gates
- Output gate: 
- Variable gates: 
- Constant gates: 
- Internal gates:
Set circuits

- Directed acyclic graph of gates
- Output gate: $\bigcirc$
- Variable gates: $x$
- Constant gates: $\top, \bot$
- Internal gates: $\times, \cup$

Factorized database fans may find these eerily familiar
Every gate $g$ captures a set $S(g)$ of sets (called assignments)
Every gate $g$ captures a set $S(g)$ of sets (called assignments).

- **Variable** gate with label $x$: $S(g) := \{\{x\}\}$
Every gate $g$ captures a set $S(g)$ of sets (called assignments).

- **Variable** gate with label $x$: $S(g) := \{\{x\}\}$
- $\top$-gates: $S(g) = \{\{\}\}$
- $\times$-gate with children $g_1, g_2$: $S(g) := \{s_1 \cup s_2 | s_1 \in S(g_1), s_2 \in S(g_2)\}$
- $\cup$-gate with children $g_1, g_2$: $S(g) := S(g_1) \cup S(g_2)$

Arithmetic circuit aficionados may see a connection.
Semiring supporters may have recognized $\{\}$.
Every gate \( g \) captures a set \( S(g) \) of sets (called assignments):

- **Variable** gate with label \( x \): \( S(g) := \{\{x\}\} \)
- \( \top \)-gates: \( S(g) = \{\}\)  
- \( \bot \)-gates: \( S(g) = \emptyset \)
Every gate $g$ captures a set $S(g)$ of sets (called assignments)

- **Variable** gate with label $x$: $S(g) := \{\{x\}\}$
- $\top$-gates: $S(g) = \{\{\}\}$
- $\bot$-gates: $S(g) = \emptyset$
- $\times$-gate with children $g_1, g_2$: $S(g) := \{s_1 \cup s_2 \mid s_1 \in S(g_1), s_2 \in S(g_2)\}$
Semantics of set circuits

Every gate $g$ captures a set $S(g)$ of sets (called assignments)

- **Variable** gate with label $x$: $S(g) := \{\{x\}\}$
- $\top$-gates: $S(g) = \{\{\}\}$
- $\bot$-gates: $S(g) = \emptyset$
- $\times$-gate with children $g_1, g_2$:
  
  $S(g) := \{s_1 \cup s_2 \mid s_1 \in S(g_1), s_2 \in S(g_2)\}$

- $\cup$-gate with children $g_1, g_2$:
  
  $S(g) := S(g_1) \cup S(g_2)$
Semantics of set circuits

Every gate $g$ captures a set $S(g)$ of sets (called assignments):

- **Variable** gate with label $x$: $S(g) := \{\{x\}\}$
- $\top$-gates: $S(g) = \{\{}\}$
- $\perp$-gates: $S(g) = \emptyset$
- $\times$-gate with children $g_1, g_2$:
  $S(g) := \{s_1 \cup s_2 \mid s_1 \in S(g_1), s_2 \in S(g_2)\}$
- $\cup$-gate with children $g_1, g_2$:
  $S(g) := S(g_1) \cup S(g_2)$

Arithmetic circuit aficionados may see a connection.
Semantics of set circuits

Every gate $g$ captures a set $S(g)$ of sets (called assignments):

- **Variable** gate with label $x$: $S(g) := \{\{x\}\}$
- $\top$-gates: $S(g) = \{\{\}\}$
- $\bot$-gates: $S(g) = \emptyset$
- $\times$-gate with children $g_1, g_2$:
  $S(g) := \{s_1 \cup s_2 \mid s_1 \in S(g_1), s_2 \in S(g_2)\}$
- $\cup$-gate with children $g_1, g_2$:
  $S(g) := S(g_1) \cup S(g_2)$

Arithmetic circuit aficionados may see a connection. Semiring supporters may have recognized Why[$X$]
Every gate $g$ captures a set $S(g)$ of sets (called assignments)

- **Variable** gate with label $x$: $S(g) := \{\{x\}\}$
- $\top$-gates: $S(g) = \{\{\}\}$
- $\bot$-gates: $S(g) = \emptyset$
- $\times$-gate with children $g_1, g_2$: $S(g) := \{s_1 \cup s_2 \mid s_1 \in S(g_1), s_2 \in S(g_2)\}$
- $\cup$-gate with children $g_1, g_2$: $S(g) := S(g_1) \cup S(g_2)$

**Arithmetic circuit aficionados** may see a connection

**Semiring supporters** may have recognized Why[X]

**Task:** Enumerate the assignments of the set $S(g)$ captured by a gate $g$

→ E.g., for $S(g) = \{\{x\}, \{x, y\}\}$, enumerate $\{x\}$ and then $\{x, y\}$
Circuit restrictions

**d-DNNF set circuit:**

- are all **deterministic:** The inputs are **disjoint** (= no assignment is captured by two inputs)
d-DNNF set circuit:

- $\cup$ are all **deterministic**:
  - The inputs are **disjoint**
    (= no assignment is captured by two inputs)

- $\times$ are all **decomposable**:
  - The inputs are **independent**
    (= no variable $x$ has a path to two different inputs)
Theorem (A., Bourhis, Jachiet, Mengel, ICALP’17)

Given a d-DNNF set circuit $C$, we can enumerate its captured assignments with preprocessing linear in $|C|$ and delay linear in each assignment.
Main results

**Theorem (A., Bourhis, Jachiet, Mengel, ICALP’17)**

Given a $d$-DNNF set circuit $C$, we can enumerate its captured assignments with preprocessing linear in $|C|$ and delay linear in each assignment.

Also: restrict to assignments of constant size $k \in \mathbb{N}$

**Theorem**

Given a $d$-DNNF set circuit $C$, we can enumerate its captured assignments of size $\leq k$ with preprocessing linear in $|C|$ and constant delay.
Main results

Theorem (A., Bourhis, Jachiet, Mengel, ICALP’17)

Given a d-DNNF set circuit C, we can enumerate its captured assignments with preprocessing \( \text{linear in } |C| \) and delay \( \text{linear in each assignment} \)

Also: restrict to assignments of constant size \( k \in \mathbb{N} \)

Theorem

Given a d-DNNF set circuit C, we can enumerate its captured assignments of size \( \leq k \) with preprocessing \( \text{linear in } |C| \) and \( \text{constant delay} \)

But where do set circuits come from?

• Directly when doing intensional query evaluation (see later)
• From Boolean circuits: you can obtain a d-DNNF set circuit:
  • From a d-DNNF, in quadratic time (smoothing)
  • From a d-SDNNF, in linear time when allowing special gates (implicit smoothing)
Main results

Theorem (A., Bourhis, Jachiet, Mengel, ICALP’17)

Given a d-DNNF set circuit $C$, we can enumerate its captured assignments with preprocessing linear in $|C|$ and delay linear in each assignment.

Also: restrict to assignments of constant size $k \in \mathbb{N}$

Theorem

Given a d-DNNF set circuit $C$, we can enumerate its captured assignments of size $\leq k$ with preprocessing linear in $|C|$ and constant delay.

But where do set circuits come from?

- Directly when doing intensional query evaluation (see later)
Main results

**Theorem (A., Bourhis, Jachiet, Mengel, ICALP’17)**

Given a d-DNNF set circuit $C$, we can enumerate its captured assignments with preprocessing *linear in* $|C|$ and delay *linear in each assignment*

Also: restrict to assignments of constant size $k \in \mathbb{N}$

**Theorem**

Given a d-DNNF set circuit $C$, we can enumerate its captured assignments of size $\leq k$ with preprocessing *linear in* $|C|$ and *constant delay*

But where do set circuits come from?

- **Directly** when doing intensional query evaluation (see later)
- **From Boolean circuits:** you can obtain a d-DNNF set circuit:
Main results

Theorem (A., Bourhis, Jachiet, Mengel, ICALP’17)
Given a $d$-DNNF set circuit $C$, we can enumerate its captured assignments with preprocessing \textit{linear in} $|C|$ and delay \textit{linear in each assignment}

Also: restrict to assignments of constant size $k \in \mathbb{N}$

Theorem
Given a $d$-DNNF set circuit $C$, we can enumerate its captured assignments of size $\leq k$ with preprocessing \textit{linear in} $|C|$ and \textit{constant delay}

But where do set circuits come from?

- Directly when doing intensional query evaluation (see later)
- From Boolean circuits: you can obtain a $d$-DNNF set circuit:
  - From a $d$-DNNF, in \textit{quadratic time} (smoothing)
Theorem (A., Bourhis, Jachiet, Mengel, ICALP’17)

Given a d-DNNF set circuit $C$, we can enumerate its captured assignments with preprocessing linear in $|C|$ and delay linear in each assignment

Also: restrict to assignments of constant size $k \in \mathbb{N}$

Theorem

Given a d-DNNF set circuit $C$, we can enumerate its captured assignments of size $\leq k$ with preprocessing linear in $|C|$ and constant delay

But where do set circuits come from?

- **Directly** when doing intensional query evaluation (see later)
- **From Boolean circuits:** you can obtain a d-DNNF set circuit:
  - From a d-DNNF, in quadratic time (smoothing)
  - From a d-SDNNF, in linear time when allowing special gates (implicit smoothing)
Proof techniques
Proof overview

Preprocessing phase:

\[ x \quad \cup \quad \times \quad x \quad z \quad d\text{-DNNF} \quad \text{set circuit} \]

Normalization (linear-time)

\[ x \quad \times \quad x \quad z \quad \text{Normalized circuit} \]

Indexing (linear-time)

\[ x \quad \times \quad x \quad z \quad \text{Indexed normalized circuit} \]

Enumeration phase:

\[ \text{Indexed normalized circuit} \quad \text{Enumeration (linear delay in each result)} \]

Results

\[ A \quad B \quad C \quad a \quad b \quad c \quad a \quad b' \quad c \]

\[ 11/24 \]
Proof overview

Preprocessing phase:

- d-DNNF set circuit
- $\rightarrow$ Normalization (linear-time)
- $\rightarrow$ Normalized circuit

Enumeration phase:

- Indexed normalized circuit
- Enumeration (linear delay in each result)
Proof overview

Preprocessing phase:

- d-DNNF set circuit
- Normalization (linear-time)
- Normalized circuit
- Indexing (linear-time)
- Indexed normalized circuit

Results: 11/24
Proof overview

**Preprocessing phase:**
- d-DNNF set circuit
- Normalization (linear-time) → Normalized circuit
- Indexing (linear-time) → Indexed normalized circuit

**Enumeration phase:**
- Indexed normalized circuit
Proof overview

Preprocessing phase:
- d-DNNF set circuit
- Normalization (linear-time)
- Normalized circuit
- Indexing (linear-time)
- Indexed normalized circuit

Enumeration phase:
- Indexed normalized circuit
- Enumeration (linear delay in each result)
- Results

\begin{align*}
\begin{array}{c}
\text{A} \\
\text{B} \\
\text{C}
\end{array} & \begin{array}{c}
\text{a} \\
\text{b} \\
\text{c}
\end{array} & \begin{array}{c}
\text{a' b' c}
\end{array}
\end{align*}
Enumerating captured assignments of d-DNNF set circuits

**Task:** Enumerate the assignments of the set $S(g)$ captured by a gate $g$

→ E.g., for $S(g) = \{\{x\}, \{x, y\}\}$, enumerate $\{x\}$ and then $\{x, y\}$
Enumerating captured assignments of d-DNNF set circuits

**Task:** Enumerate the assignments of the set $S(g)$ captured by a gate $g$

→ E.g., for $S(g) = \{\{x\}, \{x, y\}\}$, enumerate $\{x\}$ and then $\{x, y\}$

**Base case:** variable $x$: 
Task: Enumerate the assignments of the set $S(g)$ captured by a gate $g$

→ E.g., for $S(g) = \{{x}, {x, y}\}$, enumerate $\{x\}$ and then $\{x, y\}$

Base case: variable $x$: enumerate $\{x\}$ and stop
Enumerating captured assignments of d-DNNF set circuits

**Task:** Enumerate the assignments of the set $S(g)$ captured by a gate $g$

→ E.g., for $S(g) = \{\{x\}, \{x, y\}\}$, enumerate $\{x\}$ and then $\{x, y\}$

**Base case:** variable $x$ : enumerate $\{x\}$ and stop

**Concatenation:** enumerate $S(g)$ and then enumerate $S(g')$
Enumerating captured assignments of d-DNNF set circuits

Task: Enumerate the assignments of the set $S(g)$ captured by a gate $g$

→ E.g., for $S(g) = \{\{x\}, \{x, y\}\}$, enumerate $\{x\}$ and then $\{x, y\}$

Base case: variable $x$: enumerate $\{x\}$ and stop

Concatenation: enumerate $S(g)$ and then enumerate $S(g')$

Determinism: no duplicates
Enumerating captured assignments of d-DNNF set circuits

**Task:** Enumerate the assignments of the set $S(g)$ captured by a gate $g$

→ E.g., for $S(g) = \{\{x\}, \{x, y\}\}$, enumerate $\{x\}$ and then $\{x, y\}$

**Base case:** variable $x$: enumerate $\{x\}$ and stop

- **∪-gate**
  - Concatenation: enumerate $S(g)$ and then enumerate $S(g')$

- **×-gate**
  - Lexicographic product: enumerate $S(g)$ and for each result $t$ enumerate $S(g')$ and concatenate $t$ with each result

**Determinism:** no duplicates
Enumerating captured assignments of d-DNNF set circuits

**Task:** Enumerate the assignments of the set $S(g)$ captured by a gate $g$

→ E.g., for $S(g) = \{\{x\}, \{x, y\}\}$, enumerate $\{x\}$ and then $\{x, y\}$

**Base case:** variable $x$: enumerate $\{x\}$ and stop

**Concatenation:** enumerate $S(g)$ and then enumerate $S(g')$

**Determinism:** no duplicates

**Lexicographic product:** enumerate $S(g)$ and for each result $t$ enumerate $S(g')$ and concatenate $t$ with each result

**Decomposability:** no duplicates
Normalization: handling $\emptyset$

- Problem: if $S(g) = \emptyset$ we waste time
- Solution: in preprocessing compute bottom-up if $S(g) = \emptyset$ then get rid of the gate
Normalization: handling $\emptyset$

Problem: if $S(g) = \emptyset$ we waste time

Solution: in preprocessing
- compute bottom-up
- if $S(g) = \emptyset$ then get rid of the gate
Problem: if \( S(g) = \emptyset \) we waste time

Solution: in preprocessing

- compute bottom-up
- if \( S(g) = \emptyset \) then get rid of the gate
Normalization: handling $\emptyset$

Problem: if $S(g) = \emptyset$ we waste time

Solution: in preprocessing
- compute bottom-up
- if $S(g) = \emptyset$ then get rid of the gate
Normalization: handling $\emptyset$

- **Problem:** if $S(g) = \emptyset$ we waste time
• **Problem:** if $S(g) = \emptyset$ we waste time

• **Solution:** in preprocessing
  - compute **bottom-up** if $S(g) = \emptyset$
Normalization: handling $\emptyset$

- **Problem:** if $S(g) = \emptyset$ we waste time
- **Solution:** in preprocessing
  - compute **bottom-up** if $S(g) = \emptyset$
  - then get rid of the gate
Normalization: handling empty assignments

Problem: if $S(g)$ contains {} we waste time in chains of $\times$-gates

Solution:
- split $g$ between $S(g) \cap \{\}\setminus\{\}$ and $S(g) \setminus \{\}$ (homogenization)
- remove inputs with $S(g) = \{}$ for $\times$-gates
- collapse $\times$-chains with fan-in 1

Now, when traversing a $\times$-gate we make progress: non-trivial split of each set
Normalization: handling empty assignments

- **Problem:**
  - If $S(g)$ contains $\emptyset$, we waste time in chains of $\times$-gates.

- **Solution:**
  - Split $g$ between $S(g) \cap \emptyset$ and $S(g) \setminus \emptyset$ (homogenization).
  - Remove inputs with $S(g) = \emptyset$ for $\times$-gates.
  - Collapse $\times$-chains with fan-in 1.

- Now, when traversing a $\times$-gate, we make progress with a non-trivial split of each set.
Normalization: handling empty assignments

Problem:
if $S(g)$ contains $\emptyset$ we waste time in chains of $\times$-gates

Solution:
• split $g$ between $S(g) \cap \emptyset$ and $S(g) \setminus \emptyset$ (homogenization)
• remove inputs with $S(g) = \emptyset$ for $\times$-gates
• collapse $\times$-chains with fan-in 1

Now, when traversing a $\times$-gate we make progress: non-trivial split of each set
Normalization: handling empty assignments

Problem:
If $S(g)$ contains $\emptyset$ we waste time in chains of $\times$-gates

Solution:
• Split $g$ between $S(g) \cap \emptyset$ and $S(g) \setminus \emptyset$ (homogenization)
• Remove inputs with $S(g) = \emptyset$ for $\times$-gates
• Collapse $\times$-chains with fan-in 1

Now, when traversing a $\times$-gate we make progress: non-trivial split of each set.
Normalization: handling empty assignments

Problem: if $S(g)$ contains $\emptyset$ we waste time in chains of $\times$-gates

Solution:

• split $g$ between $S(g) \cap \emptyset$ and $S(g) \setminus \emptyset$ (homogenization)
• remove inputs with $S(g) = \emptyset$ for $\times$-gates
• collapse $\times$-chains with fan-in 1

→ Now, when traversing a $\times$-gate we make progress: non-trivial split of each set
Normalization: handling empty assignments

- **Problem:** if $S(g)$ contains $\emptyset$ we waste time in chains of $\times$-gates

- **Solution:**
  - split $g$ between $S(g) \cap \emptyset$ and $S(g) \setminus \emptyset$ (homogenization)
  - remove inputs with $S(g) = \emptyset$ for $\times$-gates
  - collapse $\times$-chains with fan-in 1

Now, when traversing a $\times$-gate we make progress: non-trivial split of each set
Normalization: handling empty assignments

- **Problem:** if $S(g)$ contains {} we waste time in chains of $\times$-gates

- **Solution:**
• **Problem:** if $S(g)$ contains $\emptyset$ we waste time in chains of $\times$-gates

• **Solution:**
  - split $g$ between $S(g) \cap \{\emptyset\}$ and $S(g) \setminus \{\emptyset\}$ (homogenization)
Normalization: handling empty assignments

- **Problem:** If $S(g)$ contains $\emptyset$ we waste time in chains of $\times$-gates

- **Solution:**
  - split $g$ between $S(g) \cap \emptyset$ and $S(g) \setminus \emptyset$ (homogenization)
  - remove inputs with $S(g) = \emptyset$ for $\times$-gates
Normalization: handling empty assignments

- **Problem:** if $S(g)$ contains $\emptyset$ we waste time in chains of $\times$-gates

- **Solution:**
  - split $g$ between $S(g) \cap \emptyset$ and $S(g) \setminus \emptyset$ (homogenization)
  - remove inputs with $S(g) = \emptyset$ for $\times$-gates

$$\begin{array}{c}
\{\{x\}\} \\
\times \\
\{\{x\}\} \\
\times \\
\{\{x\}\} \\
x
\end{array}$$
Normalization: handling empty assignments

- **Problem:** if $S(g)$ contains $\{\}$ we waste time in chains of $\times$-gates

- **Solution:**
  
  - split $g$ between $S(g) \cap \{\}$ and $S(g) \setminus \{\}$ (homogenization)
  - remove inputs with $S(g) = \{\}$ for $\times$-gates
  - collapse $\times$-chains with fan-in 1
Normalization: handling empty assignments

- **Problem:** if $S(g)$ contains $\emptyset$ we waste time in chains of $\times$-gates

- **Solution:**
  - split $g$ between $S(g) \cap \emptyset$ and $S(g) \setminus \emptyset$ (homogenization)
  - remove inputs with $S(g) = \emptyset$ for $\times$-gates
  - collapse $\times$-chains with fan-in 1
Normalization: handling empty assignments

• **Problem:** if $S(g)$ contains $\emptyset$ we waste time in chains of $\times$-gates

• **Solution:**
  - split $g$ between $S(g) \cap \emptyset$ and $S(g) \setminus \emptyset$ (homogenization)
  - remove inputs with $S(g) = \emptyset$ for $\times$-gates
  - collapse $\times$-chains with fan-in 1

→ Now, when traversing a $\times$-gate we make progress: non-trivial split of each set
Indexing: handling $\cup$-hierarchies

- **Problem:** we waste time in $\cup$-hierarchies to find a **reachable exit** (non-$\cup$ gate)

- **Solution:** compute reachability index

- **Problem:** must be done in linear time

- **Solution:** Determinism ensures we have a multitree (we cannot have the pattern at the right)
Indexing: handling $\bigcup$-hierarchies

- **Problem:** we waste time in $\bigcup$-hierarchies to find a **reachable exit** (non-$\bigcup$ gate)
- **Solution:** compute **reachability index**
Indexing: handling $\cup$-hierarchies

- **Problem:** we waste time in $\cup$-hierarchies to find a **reachable exit** (non-$\cup$ gate)
- **Solution:** compute **reachability index**
• **Problem:** we waste time in $\cup$-hierarchies to find a **reachable exit** (non-$\cup$ gate)

• **Solution:** compute **reachability index**

• **Problem:** must be done in **linear time**
Indexing: handling $\cup$-hierarchies

- **Problem:** we waste time in $\cup$-hierarchies to find a **reachable exit** (non-$\cup$ gate)
- **Solution:** compute **reachability index**
- **Problem:** must be done in **linear time**

- **Solution:** Determinism ensures we have a **multitree** (we cannot have the pattern at the right)
Indexing: handling $\cup$-hierarchies

- **Problem:** we waste time in $\cup$-hierarchies to find a **reachable exit** (non-$\cup$ gate)
- **Solution:** compute **reachability index**
- **Problem:** must be done in **linear time**

- **Solution:** Determinism ensures we have a **multitree** (we cannot have the pattern at the right)
- **Custom** constant-delay reachability index for multitrees
Applications
Application 1: MSO query evaluation on trees

Data: a tree $T$ where nodes have a color from an alphabet

$\begin{align*}
\text{Data: a tree } T \text{ where nodes have a color from an alphabet} & \\
\quad & \\
\end{align*}$

Query $Q$ in monadic second-order logic (MSO)

- $P(x)$ means "$x$ is blue"
- $x \rightarrow y$ means "$x$ is the parent of $y$"

"Find the pairs of a pink node and a blue node?"

$Q(x, y) := P(x) \land P(y)$

Result:

Enumerate all pairs $(a, b)$ of nodes of $T$ such that $Q(a, b)$ holds

Results:

- $(2, 7)$
- $(3, 7)$

Data complexity:

Measure efficiency as a function of $T$ (the query $Q$ is fixed)
Application 1: MSO query evaluation on trees

**Data:** a tree $T$ where nodes have a color from an alphabet

**Query $Q$** in monadic second-order logic (MSO)

- $P_\circ(x)$ means "$x$ is blue"
- $x \rightarrow y$ means "$x$ is the parent of $y$"

"Find the pairs of a pink node and a blue node?"

$Q(x, y) := P_\circ(x) \land P_\circ(y)$
Application 1: MSO query evaluation on trees

**Data:** a tree $T$ where nodes have a color from an alphabet 🌈

**Query $Q$** in monadic second-order logic (MSO)

- $P_\circ(x)$ means “$x$ is blue”
- $x \rightarrow y$ means “$x$ is the parent of $y$”

**Result:** Enumerate all pairs $(a, b)$ of nodes of $T$ such that $Q(a, b)$ holds

“Find the pairs of a pink node and a blue node?”

$Q(x, y) : = P_\circ(x) \land P_\circ(y)$

results: $(2, 7), (3, 7)$
**Application 1: MSO query evaluation on trees**

**Data:** a tree $T$ where nodes have a color from an alphabet

**Query $Q$** in monadic second-order logic (MSO)
- $P(x)$ means “$x$ is blue”
- $x \rightarrow y$ means “$x$ is the parent of $y$”

**Result:** Enumerate all pairs $(a, b)$ of nodes of $T$ such that $Q(a, b)$ holds

Querry:

```
“Find the pairs of a pink node and a blue node?”
```

$Q(x, y) := P(x) \land P(y)$

Results:

$(2, 7), (3, 7)$

**Data complexity:** Measure efficiency as a function of $T$ (the query $Q$ is fixed)
Theorem [Bagan, 2006, Kazana and Segoufin, 2013]

We can enumerate the answers of MSO queries on trees with linear-time preprocessing and constant delay.
Theorem [Bagan, 2006, Kazana and Segoufin, 2013]
We can enumerate the answers of MSO queries on trees with **linear-time preprocessing** and **constant delay**.

We can prove this with our methods:

**Theorem (A., Bourhis, Jachiet, Mengel, ICALP’17)**
For any bottom-up deterministic tree automaton $A$ and input tree $T$, we can build a **d-DNNF set circuit** capturing the results of $A$ on $T$ in $O(|A| \times |T|)$
Theorem [Bagan, 2006, Kazana and Segoufin, 2013]
We can enumerate the answers of MSO queries on trees with linear-time preprocessing and constant delay.

We can prove this with our methods:

Theorem (A., Bourhis, Jachiet, Mengel, ICALP’17)
For any bottom-up deterministic tree automaton $A$ and input tree $T$, we can build a d-DNNF set circuit capturing the results of $A$ on $T$ in $O(|A| \times |T|)$

- Can be extended to support relabeling updates to the tree in $O(\log n)$ time (A., Bourhis, Mengel, ICDT’18)
Application 1: Results

Theorem [Bagan, 2006, Kazana and Segoufin, 2013]
We can enumerate the answers of MSO queries on trees with linear-time preprocessing and constant delay.

We can prove this with our methods:

Theorem (A., Bourhis, Jachiet, Mengel, ICALP’17)
For any bottom-up deterministic tree automaton $A$ and input tree $T$, we can build a d-DNNF set circuit capturing the results of $A$ on $T$ in $O(|A| \times |T|)$

• Can be extended to support relabeling updates to the tree in $O(\log n)$ time (A., Bourhis, Mengel, ICDT’18)
• Same result for leaf insertion/deletion (A., Bourhis, Mengel, Niewerth, PODS’19) up to fixing a buggy result [Niewerth, 2018]
Application 2: Enumerating matches of nondeterministic document spanners

Data: a text $T$

Antoine Amarilli
Description Name Antoine Amarilli. Handle: a3nm. Identity Born 1990-02-07. French national. Appearance as of 2017. Auth OpenPGP. OpenId. Bitcoin. Contact Email and XMPP a3nm@a3nm.net Affiliation Associate professor of computer science (office C201-4) in the DIG team of Télécom ParisTech, 46 rue Barrault, F-75634 Paris Cedex 13, France. Studies PhD in computer science awarded by Télécom ParisTech on March 14, 2016. Former student of the École normale supérieure. test@example.com More Résumé Location Other sites Blogging: a3nm.net/blog Git: a3nm.net/git ...

Query: a pattern $P$ given as a regular expression

Output: the list of substrings of $T$ that match $P$:

$[186, 200) \cup [483, 500) \cup \ldots$

Goal:
• be very efficient in $T$ (constant-delay)
• be reasonably efficient in $P$ (polynomial-time)
Application 2: Enumerating matches of nondeterministic document spanners

Data: a text $T$

Query: a pattern $P$ given as a regular expression

$$P := \square [a-z0-9.]* \odot [a-z0-9.]* \square$$
Application 2: Enumerating matches of nondeterministic document spanners

Data: a text $T$

Antoine Amarilli Description Name Antoine Amarilli. Handle: a3nm. Identity Born 1990-02-07. French national. Appearance as of 2017. Auth OpenPGP. OpenId. Bitcoin. Contact Email and XMPP a3nm@a3nm.net Affiliation Associate professor of computer science (office C201-4) in the DIG team of Télécom ParisTech, 46 rue Barrault, F-75634 Paris Cedex 13, France. Studies PhD in computer science awarded by Télécom ParisTech on March 14, 2016. Former student of the École normale supérieure. test@example.com More Résumé Location Other sites Blogging: a3nm.net/blog Git: a3nm.net/git ...

Query: a pattern $P$ given as a regular expression

$$P := \sqcup [a-z0-9.]* \circ [a-z0-9.]* \sqcup$$

Output: the list of substrings of $T$ that match $P$:

$$[186, 200\rangle, \ [483, 500\rangle, \ ...$$
Application 2: Enumerating matches of nondeterministic document spanners

Data: a text $T$

Antoine Amarilli
Description
Name: Antoine Amarilli
Handle: a3nm
Identity
Born 1990-02-07
French national
Appearance as of 2017
Auth
OpenPGP
OpenId
Bitcoin
Contact
Email and XMPP a3nm@a3nm.net
Affiliation
Associate professor of computer science (office C201-4) in the DIG team of Télécom ParisTech, 46 rue Barrault, F-75634 Paris Cedex 13, France
Studies
PhD in computer science awarded by Télécom ParisTech on March 14, 2016
Former student of the École normale supérieure

test@example.com
More Résumé
Location
Other sites
Blogging: a3nm.net/blog
Git: a3nm.net/git ...

Query: a pattern $P$ given as a regular expression

$$P := \cup [a-z0-9.]^* @ [a-z0-9.]^* \cup$$

Output: the list of substrings of $T$ that match $P$:

$[186, 200], \ [483, 500], \ ...$

Goal:

- be very efficient in $T$ (constant-delay)
- be reasonably efficient in $P$ (polynomial-time)
Theorem (A., Bourhis, Mengel, Niewerth, ICDT’19)

We can enumerate all matches of an input nondeterministic automaton with captures on an input text with

- Preprocessing linear in the text and polynomial in the automaton
- Delay constant in the text and polynomial in the automaton
Theorem (A., Bourhis, Mengel, Niewerth, ICDT’19)

We can enumerate all matches of an input non-deterministic automaton with captures on an input text with

- Preprocessing linear in the text and polynomial in the automaton
- Delay constant in the text and polynomial in the automaton

→ Generalizes earlier result on deterministic automata [Florenzano et al., 2018]
Theorem (A., Bourhis, Mengel, Niewerth, ICDT’19)

We can enumerate all matches of an input non-deterministic automaton with captures on an input text with

- Preprocessing \textit{linear} in the text and \textit{polynomial} in the automaton
- Delay \textit{constant} in the text and \textit{polynomial} in the automaton

→ Generalizes earlier result on \textit{deterministic automata} [Florenzano et al., 2018]
- Does not really use \textit{d-DNNFs}, but \textit{bounded-width nOBDDs}
Theorem (A., Bourhis, Mengel, Niewerth, ICDT’19)
We can enumerate all matches of an input nondeterministic automaton with captures on an input text with

- Preprocessing linear in the text and polynomial in the automaton
- Delay constant in the text and polynomial in the automaton

→ Generalizes earlier result on deterministic automata [Florenzano et al., 2018]
- Does not really use d-DNNFs, but bounded-width nOBDDs
→ Generalizes to trees with polynomial dependency in the tree automaton
Application 3: Enumerating matches of annotated grammars

Data: a text $T$, e.g., source code

```c
long elt, prev, elt2, prev2=-1;
int ret = fscanf(fi, "%ld%ld", &elt, &prev);
if (ret != 2) {
    fprintf(stderr, "Bad offsets after position %ld in index!\n", pi);
    exit(1);
}
```
**Application 3: Enumerating matches of annotated grammars**

**Data:** a text $T$, e.g., source code

```c
long elt, prev, elt2, prev2=-1;
int ret = fscanf(fi, "%ld%ld", &elt, &prev);
if (ret != 2) {
    fprintf(stderr, "Bad offsets after position %ld in index!\n", pi);
    exit(1);
}
```

**Query:** a pattern $P$ given as a context-free grammar with annotated terminals

$P := \text{“find all quoted strings in the program”}$
Application 3: Enumerating matches of annotated grammars

Data: a text $T$, e.g., source code

```c
long elt, prev, elt2, prev2=-1;
int ret = fscanf(fi, "%ld%ld", &elt, &prev);
if (ret != 2) {
    fprintf(stderr, "Bad offsets after position %ld in index!\n", pi);
    exit(1);
}
```

Query: a pattern $P$ given as a context-free grammar with annotated terminals

$P := \text{“find all quoted strings in the program”}$

Theorem (A., Jachiet, Muñoz, Riveros, PODS’22)

Given an unambiguous annotation grammar $G$ and input text $w$, we can enumerate the matches with preprocessing $O(|G| \times |w|^3)$ and delay linear in each assignment
Application 3: Enumerating matches of annotated grammars

Data: a text $T$, e.g., source code

```c
long elt, prev, elt2, prev2=-1;
int ret = fscanf(fi, "%ld%ld", &elt, &prev);
if (ret != 2) {
    fprintf(stderr, "Bad offsets after position %ld in index!\n", pi);
    exit(1);
}
```

Query: a pattern $P$ given as a context-free grammar with annotated terminals

$P := \text{"find all quoted strings in the program"}$

Theorem (A., Jachiet, Muñoz, Riveros, PODS’22)

Given an unambiguous annotation grammar $G$ and input text $w$, we can enumerate the matches with preprocessing $O(|G| \times |w|^3)$ and delay linear in each assignment.

- Improves on an earlier quintic preprocessing result [Peterfreund, 2021]
Application 3: Enumerating matches of annotated grammars

Data: a text $T$, e.g., source code

```c
long elt, prev, elt2, prev2=-1;
int ret = fscanf(fi, "%ld%ld", &elt, &prev);
if (ret != 2) {
    fprintf(stderr, "Bad offsets after position %ld in index!\n", pi);
    exit(1);
}
```

Query: a pattern $P$ given as a context-free grammar with annotated terminals

$P := \text{“find all quoted strings in the program”}$

Theorem (A., Jachiet, Muñoz, Riveros, PODS’22)

Given an unambiguous annotation grammar $G$ and input text $w$, we can enumerate the matches with preprocessing $O(|G| \times |w|^3)$ and delay linear in each assignment

- Improves on an earlier quintic preprocessing result [Peterfreund, 2021]
- Quadratic and linear preprocessing for subclasses (rigid grammars, deterministic pushdown annotators)
Other applications

- Using **enumerable compact sets**, a fully-persistent version of enumerable d-DNNFs:
  - For **visibly pushdown transducers** on **nested documents** in a streaming setting [Muñoz and Riveros, 2022]
  - For **annotated automata** on **SLP-compressed documents**, with updates [Muñoz and Riveros, 2023]

- Query evaluation beyond MSO and variants on words and trees:
  - For **first-order queries** on **bounded expansion databases** [Toruńczyk, 2020]
  - For **ranked direct access** for some **CQs** with negation, see Florent’s talk this afternoon

- Can also be used to enumerate **homomorphisms between structures** [Berkholz and Vinall-Smeeth, 2023]
Other applications

• Using **enumerable compact sets**, a fully-persistent version of enumerable d-DNNFs:
  • For **visibly pushdown transducers** on **nested documents** in a streaming setting [Muñoz and Riveros, 2022]
  • For **annotated automata** on **SLP-compressed documents**, with updates [Muñoz and Riveros, 2023]

• Query evaluation beyond MSO and variants on words and trees:
  • For **first-order queries** on **bounded expansion** databases [Toruńczyk, 2020]
  • For **ranked direct access** for some **CQs with negation**, see Florent’s talk this afternoon
Other applications

- Using **enumerable compact sets**, a fully-persistent version of enumerable d-DNNFs:
  - For **visibly pushdown transducers** on **nested documents** in a streaming setting [Muñoz and Riveros, 2022]
  - For **annotated automata** on **SLP-compressed documents**, with updates [Muñoz and Riveros, 2023]

- Query evaluation beyond MSO and variants on words and trees:
  - For **first-order queries** on **bounded expansion** databases [Toruńczyk, 2020]
  - For **ranked direct access** for some **CQs with negation**, see Florent’s talk this afternoon

- Can also be used to enumerate **homomorphisms between structures** [Berkholz and Vinall-Smeeth, 2023]
What about ranked enumeration?

Enumeration algorithms typically give results in an *arbitrary (non-controllable) order*!
What about ranked enumeration?

Enumeration algorithms typically give results in an **arbitrary (non-controllable) order**!

- For **MSO queries**, ranked enumeration is **possible** with **logarithmic** delay:
  - First shown for queries on **words** [Bourhis et al., 2021]
  - Recent preprint (A., Bourhis, Capelli, Monet) for queries on **trees** under subset-monotone ranking functions
  - (Very) high-level idea: use one **priority queue** for each gate
What about ranked enumeration?

Enumeration algorithms typically give results in an arbitrary (non-controllable) order!

- For MSO queries, ranked enumeration is possible with logarithmic delay:
  - First shown for queries on words [Bourhis et al., 2021]
  - Recent preprint (A., Bourhis, Capelli, Monet) for queries on trees under subset-monotone ranking functions
  - (Very) high-level idea: use one priority queue for each gate

- For CQs, results for ranked access: [Tziavelis et al., 2022], [Deep et al., 2022], [Carmeli et al., 2023]
  - Also: see Florent’s talk
Conclusion
Summary and conclusion

- We can **enumerate** the captured assignments of d-DNNF set circuits
  - with preprocessing **linear** in the d-DNNF
  - in delay **linear** in each assignment
  - in **constant** delay for constant Hamming weight

- Applies to **MSO enumeration** on **words** and **trees**

- Applies to enumerate of the matches of **annotated context-free grammars** (with more expensive preprocessing)

- Can be used for **other applications**

- In particular: **incremental maintenance** under updates, **ranked enumeration**, etc.
Questions for future work

• What about negation gates?
• What can we do without determinism? (enumeration for DNNF?)
• Connect results on updates to finer bounds on incremental maintenance (A., Jachiet, Paperman, ICALP’21)
• Enumerate satisfying assignments via edits on previous results (A., Monet, STACS’23) to achieve constant delay even on linear-sized assignments
• For MSO queries: understand better the connection between automata classes and circuit classes (e.g., alternating automata, two-way automata...)
• More broadly, following the intensional approach for enumeration: classify enumeration tasks depending on the circuit class to which they can be compiled?
Questions for future work

- What about negation gates?
- What can we do without determinism? (enumeration for DNNF?)
- Connect results on updates to finer bounds on incremental maintenance (A., Jachiet, Paperman, ICALP’21)
- Enumerate satisfying assignments via edits on previous results (A., Monet, STACS’23) to achieve constant delay even on linear-sized assignments
- For MSO queries: understand better the connection between automata classes and circuit classes (e.g., alternating automata, two-way automata...)
- More broadly, following the intensional approach for enumeration: classify enumeration tasks depending on the circuit class to which they can be compiled?

Thanks for your attention!


Toruńczyk, S. (2020). *Aggregate queries on sparse databases.* In *PODS.*

• Set circuits can be seen as **factorized representations**
  → Not necessarily **well-typed**, height and/or assignment size may be **non-constant**
• **Determinism**: unions are disjoint
• **Decomposability**: no duplicate attribute names in products
• **Structuredness**: always the same decomposition of the attributes
Tree automata

Tree alphabet:

- Bottom-up deterministic tree automaton
- "Is there both a pink and a blue node?"
- States: \{\bot, B, P, \top\}
- Final states: \{\top\}
- Initial function: \{\bot\}
- Transitions (examples):
  - P \rightarrow \bot
  - P \rightarrow \top
  - B \rightarrow P
  - \bot \rightarrow \bot
  - \bot \rightarrow \bot
Tree automata

Tree alphabet: 🌸 🌸 🌺

- Bottom-up deterministic tree automaton
- “Is there both a pink and a blue node?”
Tree automata

Tree alphabet:  

- Bottom-up deterministic tree automaton
- “Is there both a pink and a blue node?”
- States: \{⊥, B, P, ⊤\}
Tree automata

Tree alphabet: [Diagram of tree with nodes labeled with different colors]

- Bottom-up deterministic tree automaton
- “Is there both a pink and a blue node?”
- States: \( \{ \bot, B, P, \top \} \)
- Final states: \( \{ \top \} \)
Tree automata

Tree alphabet: ○ ○ ○ ●

- Bottom-up deterministic tree automaton
- “Is there both a pink and a blue node?”
- States: \{⊥, B, P, ⊤\}
- Final states: \{⊤\}
- Initial function: ○ ⊥ ○ P ○ B
Tree automata

Tree alphabet:

- Bottom-up deterministic tree automaton
- “Is there both a pink and a blue node?”
- States: \( \{ \bot, B, P, \top \} \)
- Final states: \( \{ \top \} \)
- Initial function: \( \bot \)
Tree automata

Tree alphabet: $\emptyset \, \, B \, \, P \, \, \top$

- Bottom-up deterministic **tree automaton**
- “Is there both a pink and a blue node?”
- **States:** $\{\bot, B, P, \top\}$
- **Final states:** $\{\top\}$
- **Initial function:** $\bot \, P \, B$
- **Transitions** (examples):

```
\begin{array}{c}
P \quad \bot \\
\top \quad P \quad B \\
\bot \quad \bot
\end{array}
```
Bottom-up deterministic tree automaton

“Is there both a pink and a blue node?”

States: \{⊥, B, P, ⊤\}

Final states: \{⊤\}

Initial function:

Transitions (examples):
• Bottom-up deterministic tree automaton
• “Is there both a pink and a blue node?”
• States: \{⊥, B, P, ⊤\}
• Final states: \{⊤\}
• Initial function: \( \begin{array}{c}
∅ \quad ⊥ \\
P \\
\end{array} \quad \begin{array}{c}
P \\
B \\
\end{array} \quad ⊤ \)
• Transitions (examples):

```
  P  ⊥  P  B  ⊥  ⊥
  P  ⊥  P  B  ⊥  ⊥
  P  ⊥  P  B  ⊥  ⊥
  P  ⊥  P  B  ⊥  ⊥
  P  ⊥  P  B  ⊥  ⊥
```

Tree alphabet:
Uncertain trees

Now: Boolean query on a tree where the color of nodes is uncertain

A valuation of a tree decides whether to keep (1) or discard (0) node labels.

Valuation: \{2, 3, 7\}

\[7 \rightarrow 1, \ast 7 \rightarrow 0\]

A: “Is there both a pink and a blue node?”
Now: Boolean query on a tree where the color of nodes is **uncertain**

A **valuation** of a tree decides whether to **keep** (1) or **discard** (0) node labels.
Now: Boolean query on a tree where the color of nodes is uncertain

A valuation of a tree decides whether to keep (1) or discard (0) node labels

Valuation: \{2, 3, 7 \mapsto 1, \ast \mapsto 0\}
Uncertain trees

Now: Boolean query on a tree where the color of nodes is **uncertain**

A **valuation** of a tree decides whether to **keep** (1) or **discard** (0) node labels

**Valuation:** \{2 \mapsto 1, \ *, \mapsto 0\}
Now: Boolean query on a tree where the color of nodes is uncertain

A valuation of a tree decides whether to keep (1) or discard (0) node labels

Valuation: \(\{2, 7 \mapsto 1, \ast \mapsto 0\}\)
Now: Boolean query on a tree where the color of nodes is uncertain

A valuation of a tree decides whether to keep (1) or discard (0) node labels

Valuation: \{2, 7 \mapsto 1, \ast \mapsto 0\}

A: “Is there both a pink and a blue node?”
Now: Boolean query on a tree where the color of nodes is **uncertain**

A **valuation** of a tree decides whether to keep (1) or discard (0) node labels

Valuation: \( \{2, 3, 7 \mapsto 1, \, \ast \mapsto 0\} \)

A: “Is there both a pink and a blue node?”

The tree automaton A **accepts**
Now: Boolean query on a tree where the color of nodes is uncertain

A valuation of a tree decides whether to keep (1) or discard (0) node labels

Valuation: \{2 \mapsto 1, \ast \mapsto 0\}

A: “Is there both a pink and a blue node?”

The tree automaton A rejects
Now: Boolean query on a tree where the color of nodes is uncertain

A valuation of a tree decides whether to keep (1) or discard (0) node labels

Valuation: \{2, 7 \mapsto 1, \ast \mapsto 0\}

A: “Is there both a pink and a blue node?”

The tree automaton A accepts
Set circuit:

- Tree automaton $A$, uncertain tree $T$, circuit $C$
- **Variable gates** of $C$: nodes of $T$

Query: Is there both a pink and a blue node?
Set circuit:

- Tree automaton $A$, uncertain tree $T$, circuit $C$
- Variable gates of $C$: nodes of $T$
- Condition: Let $\nu$ be a valuation of $T$, then $A$ accepts $\nu(T)$ iff the set $S(g_0)$ of the output gate $g_0$ contains
  $$\{g \in C \mid \nu(g) = 1\}.$$
Set circuit:

- Tree automaton $A$, uncertain tree $T$, circuit $C$
- **Variable gates** of $C$: nodes of $T$
- **Condition**: Let $\nu$ be a valuation of $T$, then $A$ accepts $\nu(T)$ iff the set $S(g_o)$ of the output gate $g_o$ contains $\{g \in C \mid \nu(g) = 1\}$.

Query: Is there both a pink and a blue node?
Set circuit:

- Tree automaton $A$, uncertain tree $T$, circuit $C$
- **Variable gates** of $C$: nodes of $T$
- **Condition**: Let $\nu$ be a valuation of $T$, then $A$ accepts $\nu(T)$ iff the set $S(g_o)$ of the output gate $g_o$ contains $\{g \in C \mid \nu(g) = 1\}$.

**Query**: Is there both a pink and a blue node?
Set circuit:
- Tree automaton $A$, uncertain tree $T$, circuit $C$
- Variable gates of $C$: nodes of $T$
- Condition: Let $\nu$ be a valuation of $T$, then $A$ accepts $\nu(T)$ iff the set $S(g_0)$ of the output gate $g_0$ contains $\{g \in C \mid \nu(g) = 1\}$.

Query: Is there both a pink and a blue node?
Building provenance circuits on trees

**Theorem**

For any bottom-up deterministic **tree automaton** $A$ and input **tree** $T$, we can build a **d-DNNF set circuit** of $A$ on $T$ in $O(|A| \times |T|)$.
Building provenance circuits on trees

**Theorem**
For any bottom-up deterministic tree automaton $A$ and input tree $T$, we can build a $d$-DNNF set circuit of $A$ on $T$ in $O(|A| \times |T|)$

- **Alphabet:** 
  - States: 
    - Final: $\{ \top \}$
  - States: 
    - Final: $\{ \top \}$

- **Automaton:** “Is there both a pink and a blue node?”

- **Transitions:**
Building provenance circuits on trees

Theorem
For any bottom-up deterministic tree automaton $A$ and input tree $T$, we can build a d-DNNF set circuit of $A$ on $T$ in $O(|A| \times |T|)$.

- **Alphabet:** $\bigcirc \bigcirc \textcolor{red}{\bigcirc} \bigcirc$
- **Automaton:** “Is there both a pink and a blue node?”

- **States:** $\{\bot, B, P, \top\}$
- **Final:** $\{\top\}$

- **Transitions:**

\[ n \]\[ P \downarrow \]
\[ P \downarrow \]

\[ n \]\[ P \downarrow \]
\[ P \downarrow \]
Building provenance circuits on trees

Theorem
For any bottom-up deterministic tree automaton $A$ and input tree $T$, we can build a d-DNNF set circuit of $A$ on $T$ in $O(|A| \times |T|)$.

- **Alphabet:**
  - $\bigcirc$, $\bigcirc$, $\bigcirc$

- **Automaton:** “Is there both a pink and a blue node?”

- **States:**
  - $\{\bot, B, P, \top\}$

- **Final:**
  - $\{\top\}$

- **Transitions:**
  - $\top \bot P$
Building provenance circuits on trees

**Theorem**

For any bottom-up deterministic tree automaton $A$ and input tree $T$, we can build a d-DNNF set circuit of $A$ on $T$ in $O(|A| \times |T|)$

- **Alphabet:** [Diagram showing symbols: ⊥, B, P, ⊤]
- **Automaton:** “Is there both a pink and a blue node?”
- **States:** $\{\bot, B, P, \top\}$
- **Final:** $\{\top\}$
- **Transitions:** [Diagram of tree automaton]
Building provenance circuits on trees

**Theorem**
For any bottom-up deterministic tree automaton A and input tree T, we can build a d-DNNF set circuit of A on T in $O(|A| \times |T|)$.

- **Alphabet:** $\bigcup$
- **Automaton:** “Is there both a pink and a blue node?”
- **States:** $\{\bot, B, P, \top\}$
- **Final:** $\{\top\}$
- **Transitions:**

```
\begin{array}{c}
\top & \bot & P & \top \\
\bot & B & P & \top \\
\top & B & P & \top \\
\end{array}
```
Building provenance circuits on trees

Theorem
For any bottom-up deterministic tree automaton \( A \) and input tree \( T \), we can build a \( d \)-DNNF set circuit of \( A \) on \( T \) in \( O(|A| \times |T|) \)

- Alphabet: \( \) \( \) \( \) \( \)
- Automaton: “Is there both a pink and a blue node?”
- States: \( \{ \bot, B, P, \top \} \)
- Final: \( \{ \top \} \)
- Transitions:

\[
\begin{array}{c}
\text{States:} \\
\{ \bot, B, P, \top \} \\
\text{Final:} \{ \top \}
\end{array}
\]
Theorem

For any bottom-up deterministic tree automaton $A$ and input tree $T$, we can build a d-DNNF set circuit of $A$ on $T$ in $O(|A| \times |T|)$

- Alphabet: 〇 〇 〇 〇
- Automaton: “Is there both a pink and a blue node?”
- States: $\{\perp, B, P, \top\}$
- Final: $\{\top\}$
- Transitions:
  
  ```
  ⊤  ⊥  P
  P  ⊥  P
  ⊥  B  P  ⊤
  ⊤  B  P  ⊤
  ⊤  B  P  ⊤
  ```
The set circuit of $Q$ is now a \textbf{factorized representation} which describes all the tuples that make $Q$ true.
Circuits as factorized representations of query results

→ The set circuit of \( Q \) is now a factorized representation which describes all the tuples that make \( Q \) true

Example query:

\[ Q(X_1, X_2) : P_\circ(x) \land P_\circ(y) \]
The set circuit of $Q$ is now a factorized representation which describes all the tuples that make $Q$ true.

Example query:

$Q(X_1, X_2) : P_\circ(x) \land P_\circ(y)$

Data:

```
1
2
3
```
The set circuit of $Q$ is now a factorized representation which describes all the tuples that make $Q$ true.

Example query:
$Q(X_1, X_2) : P(x) \land P(y)$

Data:

Results:

<table>
<thead>
<tr>
<th>$X_1$</th>
<th>$X_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
Circuits as factorized representations of query results

→ The set circuit of $Q$ is now a factorized representation which describes all the tuples that make $Q$ true

Example query:
$Q(X_1, X_2) : P_0(x) \land P_0(y)$

Data:

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
```

Results:

```
<table>
<thead>
<tr>
<th>X_1</th>
<th>X_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
```

Provenance circuit:
Circuits as factorized representations of query results

→ The set circuit of $Q$ is now a **factorized representation**
  which describes all the tuples that make $Q$ true

**Example query:**

$$Q(X_1, X_2) : P_0(x) \land P_0(y)$$

<table>
<thead>
<tr>
<th>Data:</th>
<th>Results:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$X_1$</td>
</tr>
<tr>
<td>2</td>
<td>$X_2$</td>
</tr>
<tr>
<td>3</td>
<td>1 2 3</td>
</tr>
</tbody>
</table>

**Provenance circuit:**

$$\times_{X_1(1)} \cup \{X_2(2), X_2(3)\}$$
Circuits as factorized representations of query results

→ The set circuit of $Q$ is now a factorized representation which describes all the tuples that make $Q$ true

Example query:

$Q(X_1, X_2) : P_\circ(x) \land P_\circ(y)$

Data:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$X_2$</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Results:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_2$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Provenance circuit:

$\{ (X_1(1), X_2(2)), (X_1(1), X_2(3)) \}$

$\{ X_2(2), X_2(3) \}$

Theorem [Bagan, 2006, Kazana and Segoufin, 2013]

We can enumerate the answers of MSO queries on trees with linear-time preprocessing and constant delay.

Semi-open question: what about memory usage?
The set circuit of $Q$ is now a factorized representation which describes all the tuples that make $Q$ true.

Example query:

$$Q(X_1, X_2) : P_\circ(x) \land P_\circ(y)$$

Data:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$X_2$</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Results:

$$\{ (X_1(1), X_2(2)), (X_1(1), X_2(3)) \}$$

Provenance circuit:

$$\{ X_2(2), X_2(3) \}$$

Theorem [Bagan, 2006, Kazana and Segoufin, 2013]

We can enumerate the answers of MSO queries on trees with linear-time preprocessing and constant delay.
Circuits as factorized representations of query results

The set circuit of $Q$ is now a factorized representation which describes all the tuples that make $Q$ true.

Example query:

$$Q(X_1, X_2) : P(x) \land P(y)$$

Data:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Results:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X_1</td>
<td>X_2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Provenance circuit:

$$\{ (X_1(1), X_2(2)), (X_1(1), X_2(3)) \}$$

Theorem [Bagan, 2006, Kazana and Segoufin, 2013]

We can enumerate the answers of MSO queries on trees with linear-time preprocessing and constant delay.

Semi-open question: what about memory usage?