A Circuit-Based Approach to Efficient Enumeration

Antoine Amarilli\textsuperscript{1}, Pierre Bourhis\textsuperscript{2}, Louis Jachiet\textsuperscript{3}, Stefan Mengel\textsuperscript{4}

September 20th, 2017

\textsuperscript{1}Télécom ParisTech
\textsuperscript{2}CNRS CRISTAL
\textsuperscript{3}Université Grenoble-Alpes
\textsuperscript{4}CNRS CRIL
Problem statement
Problem: Enumerating large result sets

Input

Algorithm
A B C
a b c
a' b c
a b' c
a' b' c

Output
•

Problem: The output may be too large to compute efficiently

Solution: Enumerate solutions one after the other

Input
Problem: Enumerating large result sets

Input

Algorithm

Output

- Problem: The output may be too large to compute efficiently.
  
  → Solution: Enumerate solutions one after the other.
Problem: Enumerating large result sets

Input

Algorithm

Output

<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>
<tr>
<td>a'</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>a</td>
<td>b'</td>
<td>c</td>
</tr>
<tr>
<td>a'</td>
<td>b'</td>
<td>c</td>
</tr>
</tbody>
</table>

The output may be too large to compute efficiently.

Solution: Enumerate solutions one after the other.
Problem: Enumerating large result sets

- Problem: The output may be too large to compute efficiently

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a’</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>b’</td>
</tr>
<tr>
<td>a’</td>
<td>b’</td>
<td>c</td>
</tr>
</tbody>
</table>
Problem: Enumerating large result sets

- Problem: The output may be too large to compute efficiently
Problem: Enumerating large result sets

Problem: The output may be too large to compute efficiently

Algorithm

<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>
<tr>
<td>a'</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>a</td>
<td>b'</td>
<td>c</td>
</tr>
<tr>
<td>a'</td>
<td>b'</td>
<td>c</td>
</tr>
</tbody>
</table>

Output
Problem: Enumerating large result sets

- **Problem:** The output may be too large to compute efficiently.

```
<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>
<tr>
<td>a'</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>a</td>
<td>b'</td>
<td>c</td>
</tr>
<tr>
<td>a'</td>
<td>b'</td>
<td>c</td>
</tr>
</tbody>
</table>
```

Output
Problem: Enumerating large result sets

- **Problem:** The output may be too large to compute efficiently

<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>
<tr>
<td>a’</td>
<td>b</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>b’</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>a’</td>
<td>b’</td>
<td>c</td>
<td></td>
</tr>
</tbody>
</table>

Output

Results 1 - 20 of 10,514

View (previous 20 | next 20) (20 | 50 | 100 | 250 | 500)
Problem: Enumerating large result sets

- **Problem:** The output may be too large to compute efficiently

  → **Solution:** Enumerate solutions one after the other
Enumeration algorithm

Input

Step one.osf:
Indexing in O(input)

Indexed input

Step two.osf:
Enumeration in O(result)

A B C
a b c
a' b c
a b' c
a' b' c

Results

State

/three.osf//one.osf/seven.osf
Enumeration algorithm

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

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

Results

State
Enumeration algorithm

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

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

Input

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>a'</td>
</tr>
</tbody>
</table>
Enumeration algorithm

Input → Step 1: Indexing in O(input) → Indexed input → Step 2: Enumeration in O(result) → Results

State: 0011

<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 algorithm

Input → Step 1: Indexing in $O(\text{input})$ → Indexed input → Step 2: Enumeration in $O(\text{result})$ → Results

State

$0011$

<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 algorithm

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>

010001
Enumeration algorithm

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

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>

State

01100111
Enumeration algorithm

Input

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

Indexed input

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

Results

A B C

a’ b’ c

State
Enumeration algorithm

Input

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

Indexed input

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

Results

A B C

a' b' c

State
General idea for enumeration

Currently:

<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>
<tr>
<td>a</td>
<td>b'</td>
<td>c</td>
</tr>
</tbody>
</table>

Results

...
General idea for enumeration

Currently:

Input → Enumeration → Results

A B C
a b c
a b' c

Our idea:

Input → Compilation → Circuit → Enumeration → Results

\[ \lor \neg x \land z \]

A B C
a b c
a b' c

/four.osf//one.osf/seven.osf
General idea for enumeration

Currently:

Input → Enumeration → Results

![Current enumeration diagram](image)

Results

A B C
a b c
a b' c

Our idea:

Input → Compilation → Circuit → Enumeration → Results

![Our idea diagram](image)

Circuit

∨
¬
x
∧
z

A B C
a b c
a b' c

/four.osf//one.osf/seven.osf
General idea for enumeration

Currently:

Input → Enumeration → 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>
<tr>
<td>a</td>
<td>b'</td>
<td>c</td>
</tr>
</tbody>
</table>

Results

Our idea:

Input → Compilation → Circuit

Input

\[ \lor \neg x \land z \]

Circuit

\[ \lor \neg x \land z \]

Enumeration

<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>
<tr>
<td>a</td>
<td>b'</td>
<td>c</td>
</tr>
</tbody>
</table>

Enumeration

<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>
<tr>
<td>a</td>
<td>b'</td>
<td>c</td>
</tr>
</tbody>
</table>

Enumeration

<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>
<tr>
<td>a</td>
<td>b'</td>
<td>c</td>
</tr>
</tbody>
</table>
General idea for enumeration

Currently:

Input → Enumeration → Results

A B C
a b c
a b' c

Results

Our idea:

Input → Compilation → Circuit

\( \lor \neg x \land z \)

Input → Compilation → Circuit

\( \lor \neg x \land z \)

\( \lor \neg x \land z \)

\( \lor \neg x \land z \)

A B C
a b c
a b' c

Results

/four.osf//one.osf/seven.osf
General idea for enumeration

Currently:

Input → Enumeration → Results

A  B  C
a  b  c  
|  |  |
|---|---|---|
a  b'  c

Results

Our idea:

Input → Compilation → Circuit

\[ \lor \quad \neg x \quad \land z \]

Input → Compilation → Circuit

\[ \lor \quad \neg x \quad \land z \]

Input → Compilation → Circuit

\[ \lor \quad \neg x \quad \land z \]

\[ /four.osf//one.osf/seven.osf \]
General idea for enumeration

Currently:

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

Results

Our idea:

Input

Compilation

Circuit

<table>
<thead>
<tr>
<th>Input</th>
<th>Compilation</th>
<th>Circuit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>∨ ¬ x ∧ z</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>∨ ¬ x ∧ z</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>∨ ¬ x ∧ z</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Enumeration</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>∨ ¬ x ∧ z</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>∨ ¬ x ∧ z</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>∨ ¬ x ∧ z</td>
<td></td>
</tr>
</tbody>
</table>

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

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

- Directed acyclic graph of gates
- Output gate:
- Variable gates: $x$
- Internal gates: $\lor$, $\land$, $\neg$

Example valuation: $\nu = \{x \mapsto \text{zero.osf}, y \mapsto \text{one.osf}\}$

Assignment: $S\nu = \{y\}$; more concise than $\nu$

Our task: Enumerate all satisfying assignments of an input circuit
Boolean circuits

- Directed acyclic graph of gates
- Output gate:
- Variable gates:
- Internal gates:
- Valuation: function from variables to \( \{0, 1\} \)
  
Example: \( \nu = \{x \mapsto 0, \ y \mapsto 1\} \)...

\( \neg \) \( \lor \) \( \land \)
Boolean circuits

- Directed acyclic graph of gates
- Output gate: \( \bigcirc \)
- Variable gates: \( x \)
- Internal gates: \( \lor, \land, \neg \)
- Valuation: function from variables to \{ 0, 1 \}

Example: \( \nu = \{ x \mapsto 0, \ y \mapsto 1 \} \)
Boolean circuits

- Directed acyclic graph of gates
- Output gate:
- Variable gates:
- Internal gates:
- Valuation: function from variables to \{0, 1\}
  Example: \( \nu = \{ x \mapsto 0, \ y \mapsto 1 \} \)
Boolean circuits

- Directed acyclic graph of gates
- Output gate:
- Variable gates:
- Internal gates:
- Valuation: function from variables to \{0, 1\}
  Example: \(\nu = \{x \mapsto 0, \ y \mapsto 1\}\) ... mapped to 1
Boolean circuits

- Directed acyclic graph of **gates**
- **Output** gate: $\bigcirc$
- **Variable gates:** $x$
- **Internal gates:** $\lor, \land, \neg$

**Valuation:** function from variables to $\{0, 1\}$
Example: $\nu = \{x \mapsto 0, y \mapsto 1\}$... mapped to $1$

**Assignment:** set of variables mapped to $1$
Example: $S_\nu = \{y\}$; more concise than $\nu$
Boolean circuits

- Directed acyclic graph of gates
- Output gate: 
- Variable gates: 
- Internal gates: 
- Valuation: function from variables to \( \{0, 1\} \)
  Example: \( \nu = \{x \mapsto 0, y \mapsto 1\} \) ... mapped to 1
- Assignment: set of variables mapped to 1
  Example: \( S_\nu = \{y\} \); more concise than \( \nu \)

Our task: Enumerate all satisfying assignments of an input circuit
d-DNNF:

- are all deterministic:
The inputs are mutually exclusive (= no valuation \( \nu \) makes two inputs simultaneously evaluate to 1)
Circuit restrictions

**d-DNNF:**

- \( \lor \) are all **deterministic:**
  The inputs are **mutually exclusive**
  (= no valuation \( \nu \) makes two inputs simultaneously evaluate to \( 1 \))

- \( \land \) are all **decomposable:**
  The inputs are **independent**
  (= no variable \( x \) has a path to two different inputs)
Circuit restrictions

**d-DNNF:**

- ∨ are all **deterministic:**
  The inputs are **mutually exclusive** (= no valuation \( \nu \) makes two inputs simultaneously evaluate to 1)

- ∧ are all **decomposable:**
  The inputs are **independent** (= no variable \( x \) has a path to two different inputs)

**v-tree:** ∧-gates follow a tree on the variables
Main results

Theorem

Given a $d$-DNNF circuit $C$ with a v-tree $T$, we can enumerate its satisfying assignments with preprocessing linear in $|C| + |T|$ and delay linear in each assignment.
Main results

Theorem

Given a \textit{d-DNNF circuit} $C$ with a \textit{v-tree} $T$, we can enumerate its \textit{satisfying assignments} with preprocessing \textit{linear in} $|C| + |T|$ and delay \textit{linear in each assignment}

Also: restrict to assignments of \textit{constant size} $k \in \mathbb{N}$ (at most $k$ variables are set to 1):

Theorem

Given a \textit{d-DNNF circuit} $C$ with a \textit{v-tree} $T$, we can enumerate its \textit{satisfying assignments} of size $\leq k$ with preprocessing \textit{linear in} $|C| + |T|$ and \textit{constant delay}
**Application 1: Factorized databases**

<table>
<thead>
<tr>
<th>Orders (O for short)</th>
<th>Dish (D for short)</th>
<th>Items (I for short)</th>
</tr>
</thead>
<tbody>
<tr>
<td>customer</td>
<td>day</td>
<td>dish</td>
</tr>
<tr>
<td>Elise</td>
<td>Monday</td>
<td>burger</td>
</tr>
<tr>
<td>Elise</td>
<td>Friday</td>
<td>burger</td>
</tr>
<tr>
<td>Steve</td>
<td>Friday</td>
<td>hotdog</td>
</tr>
<tr>
<td>Joe</td>
<td>Friday</td>
<td>hotdog</td>
</tr>
</tbody>
</table>

Consider the join of the above relations:

\[
O(\text{customer, day, dish}), D(\text{dish, item}), I(\text{item, price})
\]

<table>
<thead>
<tr>
<th>customer</th>
<th>day</th>
<th>dish</th>
<th>item</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elise</td>
<td>Monday</td>
<td>burger</td>
<td>patty</td>
<td>6</td>
</tr>
<tr>
<td>Elise</td>
<td>Monday</td>
<td>burger</td>
<td>onion</td>
<td>2</td>
</tr>
<tr>
<td>Elise</td>
<td>Monday</td>
<td>burger</td>
<td>bun</td>
<td>2</td>
</tr>
<tr>
<td>Elise</td>
<td>Friday</td>
<td>burger</td>
<td>patty</td>
<td>6</td>
</tr>
<tr>
<td>Elise</td>
<td>Friday</td>
<td>burger</td>
<td>onion</td>
<td>2</td>
</tr>
<tr>
<td>Elise</td>
<td>Friday</td>
<td>burger</td>
<td>bun</td>
<td>2</td>
</tr>
</tbody>
</table>

(Slides courtesy of Dan Olteanu)
Application 1: Factorized databases

\(_O(\text{customer, day, dish}), D(\text{dish, item}), I(\text{item, price})\)_

<table>
<thead>
<tr>
<th>customer</th>
<th>day</th>
<th>dish</th>
<th>item</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elise</td>
<td>Monday</td>
<td>burger</td>
<td>patty</td>
<td>6</td>
</tr>
<tr>
<td>Elise</td>
<td>Monday</td>
<td>burger</td>
<td>onion</td>
<td>2</td>
</tr>
<tr>
<td>Elise</td>
<td>Monday</td>
<td>burger</td>
<td>bun</td>
<td>2</td>
</tr>
<tr>
<td>Elise</td>
<td>Friday</td>
<td>burger</td>
<td>patty</td>
<td>6</td>
</tr>
<tr>
<td>Elise</td>
<td>Friday</td>
<td>burger</td>
<td>onion</td>
<td>2</td>
</tr>
<tr>
<td>Elise</td>
<td>Friday</td>
<td>burger</td>
<td>bun</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

A relational algebra expression encoding the above query result is:

\(<\text{Elise}\> \times <\text{Monday}\> \times <\text{burger}\> \times <\text{patty}\> \times <6> \cup
\<\text{Elise}\> \times <\text{Monday}\> \times <\text{burger}\> \times <\text{onion}\> \times <2> \cup
\<\text{Elise}\> \times <\text{Monday}\> \times <\text{burger}\> \times <\text{bun}\> \times <2> \cup
\<\text{Elise}\> \times <\text{Friday}\> \times <\text{burger}\> \times <\text{patty}\> \times <6> \cup
\<\text{Elise}\> \times <\text{Friday}\> \times <\text{burger}\> \times <\text{onion}\> \times <2> \cup
\<\text{Elise}\> \times <\text{Friday}\> \times <\text{burger}\> \times <\text{bun}\> \times <2> \cup \ldots

(Slides courtesy of Dan Olteanu)
Application 1: Factorized databases

Decomposable: by definition (following the schema)

Deterministic: we do not obtain the same tuple multiple times

Theorem (Strengthens result of [Olteanu and Závodný, /two.osf/zero.osf/one.osf/five.osf])

Given a deterministic factorized representation, we can enumerate its
tuples with linear preprocessing and constant delay

(Slides courtesy of Dan Olteanu)
Application 1: Factorized databases

(Slides courtesy of Dan Olteanu)
**Application 1: Factorized databases**

- **Decomposable**: by definition (following the schema)
- **Deterministic**: we do not obtain the same tuple multiple times
Application 1: Factorized databases

- Decomposable: by definition (following the schema)
- Deterministic: we do not obtain the same tuple multiple times

**Theorem (Strenghtens result of [Olteanu and Závodný, 2015])**

Given a deterministic factorized representation, we can enumerate its tuples with *linear preprocessing* and *constant delay*
Application 2: Query evaluation

Query evaluation on trees

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

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

Result: TRUE/FALSE indicating if $T$ satisfies the query $Q$

“Is there both a pink and a blue node?”
$\exists x \, y \, P(x) \land P(y)$

Computational complexity as a function of the tree $T$
(the query $Q$ is fixed)

(Slides courtesy of Pierre Bourhis)
Application 2: Query evaluation

- Compute the results \((a, b, c)\) of a query \(Q(x, y, z)\) on a tree \(T\)
  - Generalizes to \textit{bounded-treewidth} databases

\(\quad\)

Theorem (Recaptures \cite{Bagan, Kazana and Segou})
For any constant \(k \in \mathbb{N}\) and \textit{fixed MSO query} \(Q\),
given a database \(D\) of treewidth \(\leq k\),
the results of \(Q\) on \(D\) can be enumerated with linear preprocessing in \(D\) and linear delay in each answer (\(\rightarrow\) constant delay for free \textit{first-order variables})
Application 2: Query evaluation

• Compute the results \((a, b, c)\) of a query \(Q(x, y, z)\) on a tree \(T\)
  \(\rightarrow\) Generalizes to bounded-treewidth databases

• Query given as a deterministic tree automaton
  \(\rightarrow\) Captures monadic second-order (data-independent translation)
  \(\rightarrow\) Captures conjunctive queries, SQL, etc.
Application 2: Query evaluation

• Compute the results \((a, b, c)\) of a query \(Q(x, y, z)\) on a tree \(T\)
  → Generalizes to bounded-treewidth databases

• Query given as a deterministic tree automaton
  → Captures monadic second-order (data-independent translation)
  → Captures conjunctive queries, SQL, etc.

→ We can construct a \(d\)-DNNF that describes the query results
Application 2: Query evaluation

- Compute the results \((a, b, c)\) of a query \(Q(x, y, z)\) on a tree \(T\)
  \(\rightarrow\) Generalizes to bounded-treewidth databases

- Query given as a deterministic tree automaton
  \(\rightarrow\) Captures monadic second-order (data-independent translation)
  \(\rightarrow\) Captures conjunctive queries, SQL, etc.

\(\rightarrow\) We can construct a d-DNNF that describes the query results

**Theorem (Recaptures [Bagan, 2006], [Kazana and Segoufin, 2013])**

For any constant \(k \in \mathbb{N}\) and fixed MSO query \(Q\),
given a database \(D\) of treewidth \(\leq k\), the results of \(Q\) on \(D\)
can be enumerated with linear preprocessing in \(D\) and linear delay
in each answer (\(\rightarrow\) constant delay for free first-order variables)
Proof techniques
Proof overview

Preprocessing phase:

\[ \neg x \land z \]

Circuit

\[ x \]

\[ y \land z \]

v-tree

Translation (linear-time)

\[ \lor \land x \land z \]

Circuit in zero-suppressed semantics

Normalization (linear-time)

\[ \land x \land z \]

Normalized circuit

Enumeration phase:

\[ \land x \land z \]

Enumeration (linear delay in each result)

\[ a \land b \land c \land a \land b' \land c \]

Results

\[ /one.osf/zero.osf//one.osf/seven.osf \]
Proof overview

Preprocessing phase:

Circuit

\( \neg x \wedge z \)

\( x \wedge z \)

v-tree

Translation (linear-time)

\( \neg x \wedge z \)

\( x \wedge z \)

in zero-suppressed semantics

\( a \wedge b \wedge c \)

Results

\( /one.osf/zero.osf//one.osf/seven.osf \)
Proof overview

Preprocessing phase:

\[ \neg \land x \land z \\]

Circuit

\[ \neg x \land \land z \\]

\[ y \land \land z \\]

\[ \land \land x \land z \\]

\[ \land x \land z \\]

v-tree

Translation (linear-time)

Circuit in zero-suppressed semantics

Normalization (linear-time)

Normalized circuit

\[ a \land \land b' \land \land c \\]

Results

\[ a \land \land b \land \land c \\]

\[ \land \land x \land z \\]

\[ \land x \land z \\]
Proof overview

Preprocessing phase:

Translation (linear-time)

Circuit in zero-suppressed semantics

Normalization (linear-time)

Normalized circuit

Enumeration phase:

Normalized circuit
Proof overview

Preprocessing phase:

Translation (linear-time) → Circuit in zero-suppressed semantics → Normalization (linear-time) → Normalized circuit

Circuit

v-tree

Enumeration phase:

Enumeration (linear delay in each result) → Results

Normalized circuit

<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>
<tr>
<td>a</td>
<td>b'</td>
<td>c</td>
</tr>
</tbody>
</table>
Zero-suppressed semantics

Special zero-suppressed semantics for circuits:
Zero-suppressed semantics

Special **zero-suppressed semantics** for circuits:

- No **NOT**-gate
- Each gate *captures* a set of assignments
- **Bottom-up** definition with $\times$ and $\cup$

\[\{y, \{z\}\}, \{x, \{y, \{x, \{z\}\}\}\}\]
Special zero-suppressed semantics for circuits:

- No NOT-gate
- Each gate captures a set of assignments
- Bottom-up definition with $\times$ and $\cup$

$$\{\{y\}, \{z\}\}$$
Special zero-suppressed semantics for circuits:

- No \textbf{NOT}-gate
- Each gate \textit{captures} a set of assignments
- \textbf{Bottom-up} definition with $\times$ and $\cup$

$\{\{x, y\}, \{x, z\}\}$

$\{\{y\}, \{z\}\}$
Zero-suppressed semantics

Special zero-suppressed semantics for circuits:

- No NOT-gate
- Each gate captures a set of assignments
- Bottom-up definition with $\times$ and $\cup$

- d-DNNF: $\cup$ are disjoint, $\times$ are on disjoint sets
Zero-suppressed semantics

Special zero-suppressed semantics for circuits:
- No NOT-gate
- Each gate captures a set of assignments
- Bottom-up definition with $\times$ and $\cup$

- d-DNNF: $\cup$ are disjoint, $\times$ are on disjoint sets

Many equivalent ways to understand this:
- Generalization of factorized representations
- Analogue of zero-suppressed OBDDs (implicit negation)
- Arithmetic circuits: $\times$ and $+$ on polynomials
Zero-suppressed semantics

\{\{x, y\}, \{x, z\}\}

Special \textbf{zero-suppressed semantics} for circuits:

- No \textbf{NOT}-gate
- Each gate \textbf{captures} a set of assignments
- \textbf{Bottom-up} definition with $\times$ and $\cup$

- \textbf{d-DNNF}: $\cup$ are disjoint, $\times$ are on disjoint sets

Many \textbf{equivalent ways} to understand this:

- Generalization of \textbf{factorized representations}
- Analogue of \textbf{zero-suppressed} OBDDs (implicit negation)
- \textbf{Arithmetic circuits}: $\times$ and $+$ on polynomials

\textbf{Simplification}: rewrite circuits to arity-two (fan-in $\leq 2$)
Task: Enumerate the elements of the set $S(g)$ captured by a gate $g$

→ E.g., for $S(g) = \{\{x, y\}, \{x, z\}\}$, enumerate $\{x, y\}$ and then $\{x, z\}$
Task: Enumerate the elements of the set $S(g)$ captured by a gate $g$

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

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

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

Base case: variable $x$: enumerate $\{x\}$ and stop
Task: Enumerate the elements of the set $S(g)$ captured by a gate $g$

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

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

**OR-gate**

$\bigcirc v$

\[ g \quad g' \]

**Concatenation:** enumerate $S(g)$ and then enumerate $S(g')$
Enumerating assignments in the zero-suppressed semantics

**Task:** Enumerate the elements of the set \( S(g) \) captured by a gate \( g \)

\[ \rightarrow \text{E.g., for } S(g) = \{ \{ x, y \}, \{ x, z \} \}, \text{enumerate } \{ x, y \} \text{ and then } \{ x, z \} \]

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

**OR-gate**

\[ g \quad \lor \quad g' \]

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

**Determinism:** no duplicates
Task: Enumerate the elements of the set $S(g)$ captured by a gate $g$

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

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

**OR-gate**

$$\lor$$

$g$  $g'$

**AND-gate**

$$\land$$

$g$  $g'$

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

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

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: compute bottom-up if $S(g) = \emptyset$
Normalization: handling $\emptyset$

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

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

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

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

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

Solution: compute bottom-up if $S(g) = \emptyset$
• **Problem**: if \( S(g) = \emptyset \) we waste time
Normalization: handling $\emptyset$

- **Problem:** if $S(g) = \emptyset$ we waste time
- **Solution:** compute **bottom-up** if $S(g) = \emptyset$
Normalizing: handling empty assignments

- Problem: If $S(g)$ contains $\emptyset$, we waste time in chains of AND-gates.
- Solution:
  - Split $g$ between $S(g) \cap \emptyset$ and $S(g) \setminus \emptyset$ (homogenization).
  - Remove inputs with $S(g) = \emptyset$ for AND-gates.
  - Collapse AND-chains with fan-in /one.osf/four.osf//one.osf/seven.osf.

Now, traversing an AND-gate ensures that we make progress: it splits the assignments non-trivially.
Normalization: handling empty assignments

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

• Solution:
  • split $g$ between $S(g) \cap \emptyset$ and $S(g) \setminus \emptyset$ (homogenization)
  • remove inputs with $S(g) = \emptyset$ for AND-gates
  • collapse AND-chains with fan-in /one.osf/four.osf//one.osf/seven.osf

→ Now, traversing an AND-gate ensures that we make progress: it splits the assignments non-trivially
Normalization: handling empty assignments

- Problem: if $S(g)$ contains $\emptyset$, we waste time in chains of AND-gates.
- Solution:
  - Split $g$ between $S(g) \cap \emptyset$ and $S(g) \setminus \emptyset$ (homogenization).
  - Remove inputs with $S(g) = \emptyset$ for AND-gates.
  - Collapse AND-chains with fan-in /one.osf /four.osf /seven.osf

Now, traversing an AND-gate ensures that we make progress: it splits the assignments non-trivially.
Normalization: handling empty assignments

- Problem: if $S(g)$ contains $\emptyset$ we waste time in chains of AND-gates
- Solution:
  - split $g$ between $S(g) \cap \emptyset$ and $S(g) \setminus \emptyset$ (homogenization)
  - remove inputs with $S(g) = \emptyset$ for AND-gates
  - collapse AND-chains with fan-in /one.osf/four.osf//one.osf/seven.osf

Now, traversing an AND-gate ensures that we make progress: it splits the assignments non-trivially
Normalization: handling empty assignments

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

Solution:
• split $g$ between $S(g) \cap \emptyset$ and $S(g) \setminus \emptyset$ (homogenization)
• remove inputs with $S(g) = \emptyset$ for AND-gates
• collapse AND-chains with fan-in /one.osf /four.osf //one.osf /seven.osf

Now, traversing an AND-gate ensures that we make progress: it splits the assignments non-trivially
Normalization: handling empty assignments

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

\[
\begin{align*}
\{\{x\}\} & \\
\wedge & \\
\{\emptyset\} & \{\{x\}\} & \\
\wedge & \\
\{\emptyset\} & \{\{x\}\} & \\
\wedge & \\
\{\emptyset\} & \{\{x\}\} & \\
\wedge & \\
\{\emptyset\} & \{\{x\}\} & \\
\text{x} & \\
\end{align*}
\]
Normalization: handling empty assignments

- **Problem:** if $S(g)$ contains $\emptyset$ we waste time in chains of AND-gates
- **Solution:**
Normalization: handling empty assignments

- **Problem:** if $S(g)$ contains $\{\}$ we waste time in chains of AND-gates
- **Solution:**
  - split $g$ between $S(g) \cap \{\{}\}$ and $S(g) \setminus \{\{}\}$ (homogenization)
Normalization: handling empty assignments

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

Solution:

- split $g$ between $S(g) \cap \{\}$ and $S(g) \setminus \{\}$ (homogenization)
- remove inputs with $S(g) = \{\}$ for AND-gates
Normalization: handling empty assignments

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

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

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

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

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

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

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

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

→ Now, traversing an **AND-gate** ensures that we make progress: it **splits** the assignments non-trivially
- Problem: we waste time in OR-hierarchies to find a reachable exit (non-OR gate)
Normalization: handling OR-hierarchies

- **Problem:** we waste time in OR-hierarchies to find a *reachable exit* (non-OR gate)
- **Solution:** compute *reachability index*
Normalization: handling OR-hierarchies

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

- **Solution:** compute **reachability index**

\[
\begin{align*}
    & g_1 \\
    & \lor \\
    & g_2 \\
    & \lor \\
    & g_3 \\
    & \lor \\
    & g_4 \\
    & \lor 
\end{align*}
\]
Normalization: handling OR-hierarchies

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

- **Problem:** we waste time in OR-hierarchies to find a **reachable exit** (non-OR 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
Translating to zero-suppressed semantics

- This is where we use the \( v \)-tree

\[
\begin{align*}
&x \\
&\hspace{1cm} y \\
&\hspace{2cm} z
\end{align*}
\]

Problem: quadratic blowup

Solution:

- Order \( x < y < z \)
- Interval \([x, z]\)
- Range gates to denote \( \vee [x, z] \) in constant space
Translating to zero-suppressed semantics

- This is where we use the v-tree
- Add explicitly untested variables (smoothing)
Translating to zero-suppressed semantics

- This is where we use the v-tree
- Add explicitly **untested variables** (smoothing)
Translating to zero-suppressed semantics

• This is where we use the v-tree
• Add explicitly untested variables (smoothing)

• Problem: quadratic blowup
Translating to zero-suppressed semantics

- This is where we use the v-tree
- Add explicitly **untested variables** (smoothing)

\[
\text{x} \land \text{y} \land \text{z} \lor \text{x} \land \text{y} \lor \text{y} \land \text{z} \lor \text{z} \\
\]

- Problem: quadratic blowup
- Solution:
  - **Order** < on variables in the v-tree
    \((x < y < z)\)
  - Interval \([x, z]\)
  - Range gates to denote \(\lor[x, z]\)
    in constant space
Translating to zero-suppressed semantics

- This is where we use the v-tree
- Add explicitly untested variables (smoothing)

Problem: quadratic blowup
Solution:
  - Order < on variables in the v-tree \((x < y < z)\)
  - Interval \([x, z]\)
  - Range gates to denote \(\lor[x, z]\) in constant space
Conclusion
Summary and conclusion

- **Enumerate** the satisfying assignments of structured d-DNNF
  - in delay **linear** in each assignment
  - in **constant** delay for constant Hamming weight
- Can **recapture** existing enumeration results
- Useful **general-purpose** result for applications

Future work:
- **Practice**: implement the technique with automata
- **Theory**: handle updates on the input

Thanks for your attention!
Summary and conclusion

• **Enumerate** the satisfying assignments of structured d-DNNF
  → in delay **linear** in each assignment
  → in **constant** delay for constant Hamming weight
→ Can **recapture** existing enumeration results
→ Useful **general-purpose** result for applications

Future work:

• **Practice**: implement the technique with automata
• **Theory**: handle **updates** on the input
Summary and conclusion

• **Enumerate** the satisfying assignments of structured d-DNNF
  → in delay **linear** in each assignment
  → in **constant** delay for constant Hamming weight
→ Can **recapture** existing enumeration results
→ Useful **general-purpose** result for applications

Future work:

• **Practice**: implement the technique with automata
• **Theory**: handle **updates** on the input

**Enumeration on Trees under Relabelings**

Antoine Amarilli, Pierre Bourhis, Stefan Mengel

*(Submitted on 18 Sep 2017)*
Summary and conclusion

- **Enumerate** the satisfying assignments of structured d-DNNF
  - in delay **linear** in each assignment
  - in **constant** delay for constant Hamming weight
- Can **recapture** existing enumeration results
- Useful **general-purpose** result for applications

Future work:

- **Practice**: implement the technique with automata
- **Theory**: handle **updates** on the input

arXiv.org > cs > arXiv:1709.06185

Computer Science > Databases

Enumeration on Trees under Relabelings

Antoine Amarilli, Pierre Bourhis, Stefan Mengel

(Submitted on 18 Sep 2017)

Thanks for your attention!
Bagan, G. (2006). **MSO queries on tree decomposable structures are computable with linear delay.**
In *CSL*.
