A Circuit-Based Approach to Efficient Enumeration

Antoine Amarilli¹, Pierre Bourhis², Louis Jachiet³, Stefan Mengel⁴
May 9th, 2017

¹Télécom ParisTech
²CNRS CRIStAL
³Université Grenoble-Alpes
⁴CNRS CRIL
Problem statement
Problem: Enumerating large result sets

Input

Output

• The output may be too large to compute efficiently

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

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>
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
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>
```

Results 1 - 20 of 10,514

Search
Problem: Enumerating large result sets

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

```
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>
```

Results 1 - 20 of 10,514
Problem: Enumerating large result sets

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

```
Search

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

Output:

<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>

Solution: Enumerate solutions one after the other

Results 1 - 20 of 10,514

View (previous 20 | next 20) (20 | 50 | 100 | 250 | 500)
Enumeration algorithm

Input

Step /one.osf:
Indexing in $O(\text{input})$

Step /two.osf:
Enumeration in $O(\text{result})$

Results

State: /three.osf//one.osf/three.osf
Enumeration algorithm

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

Input

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

Indexed input

ResultsState

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

Input

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

Indexed input

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

Step 1: Indexing in O(input)

Step 2: Enumeration in O(result)

Results
Enumeration algorithm

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

Indexed input

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

Results

State

Input

A  B  C
a  b  c

0011
Enumeration algorithm

Step 1: Indexing in O(input)

Step 2: Enumeration in O(result)

Input

Indexed input

Results

State

A
B
C
a
b
c

0011
Enumeration algorithm

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

State: 010001

A B C

a' b c
Enumeration algorithm

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

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

Input → Indexed input → State

Results

A B C

a b' c

01100111
Enumeration algorithm

**Input**

1. **Step 1:** Indexing in $O(\text{input})$
   - Indexed input

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

   - Results
   - State

   - $A \ B \ C$
   - $a' \ b' \ c$
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>

Result:

Our idea:

\[ \text{Input} \xrightarrow{\text{Compilation}} \bigvee \neg x \land z \xrightarrow{\text{Circuit}} \bigvee \neg x \land z \xrightarrow{\text{Enumeration}}\]

\[\text{Results} \quad a \quad b \quad c \quad a \quad b' \quad c\]
General idea for enumeration

Currently:

Input → Enumeration → Results

Input → Enumeration → Results

Our idea:

Input → Compilation → Circuit → Results

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

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

\[ \lor \neg x \land z \]
General idea for enumeration

Currently:

- **Input** → **Enumeration** → **Results**

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>

Our idea:

- **Input** → **Compilation** → **Circuit** → **Enumeration** → **Results**

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>
General idea for enumeration

Currently:

Input → \( A \ B \ C \)

\( a \ b \ c \)

\( a \ b' \ c \)

Results

Our idea:

Input → Compilation → \( \lor \neg x \land z \) → Circuit

Input → \( A \ B \ C \)

\( a \ b \ c \)

\( a \ b' \ c \)

Results

\( /four.osf//one.osf/three.osf \)
General idea for enumeration

Currently:

Input → Enumeration → Results

A B C
a b c
a b' c

Results

Our idea:

Input → Compilation → Circuit

Input → Compilation → Circuit

A B C
a b c
a b' c

¬ x ∨ z

Circuit
General idea for enumeration

Currently:

Input → Enumeration → Results

A B C
a b c
a b' c

Results

Our idea:

Input → Compilation → Circuit

A B C
a b c
a b' c

Compilation

Circuit

Input → Compilation → Circuit

A B C
a b c
a b' c

Compilation

Circuit

Input → Compilation → Circuit

A B C
a b c
a b' c

Compilation

Circuit

Input → Compilation → Circuit

A B C
a b c
a b' c

Compilation

Circuit
General idea for enumeration

Currently:

Input → Enumeration → Results

A B C
a b c a b' c

Input → Enumeration → Results

A B C
a b c a b' c

Input → Enumeration → Results

A B C
a b c a b' c

Our idea:

Input → Compilation → Circuit

\[ \lor \neg \land \]

\[ x z \]

Input → Compilation → Circuit

\[ \lor \neg \land \]

\[ x z \]

Input → Compilation → Circuit

\[ \lor \neg \land \]

\[ x z \]

Input → Enumeration → Results

A B C
a b c a b' c
Boolean circuits

- Directed acyclic graph of gates

Example:

\[ \nu = \{ x \mapsto \text{zero.osf}, y \mapsto \text{one.osf} \} \]

Assignment: set of variables mapped to \text{one.osf}

Example:

\[ S_\nu = \{ y \} \]

More concise than \nu

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

- Directed acyclic graph of gates
- Output gate: 

Example:

\[ \nu = \{ x \mapsto \text{zero.osf}, y \mapsto \text{one.osf} \} \]

Assignment: set of variables mapped to one.osf

Example:

\[ S_\nu = \{ y \} \]; 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: 

Example:

\[
\nu = \{x \mapsto \text{zero}, y \mapsto \text{one}\}
\]

Assignment: set of variables mapped to one.

Example:

\[
S\nu = \{y\}; \text{more concise than } \nu
\]

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

- Directed acyclic graph of **gates**
- **Output** gate:
- **Variable** gates: $x$, $y$
- **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: \( \nu = \{x \mapsto 0, \ y \mapsto 1\} \ldots \)
Boolean circuits

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

Our task: Enumerate all satisfying assignments of an input circuit
Circuit restrictions

**d-DNNF:**

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

![Diagram of a d-DNNF circuit]
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)
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)

**v-tree:** \( \land \)-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

- **Factorized databases**: implicit representation of database tables
Application 1: Factorized databases

- Factorized databases: implicit representation of database tables

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

represented by

\[
\{ \langle B : b \rangle, \langle B : b' \rangle \} \}
\]

\[
\times \cup \]

Relational product

Relational union

Deterministic: We do not obtain the same tuple multiple times

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

Given a deterministic factorized representation, we can enumerate its tuples with linear preprocessing and constant delay.
Application 1: Factorized databases

- **Factorized databases**: implicit representation of database tables

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

represented by

\[
\langle A : a \rangle \times \left\{ \langle B : b \rangle, \langle B : b' \rangle \right\}
\]

\[
\langle B : b \rangle \cup \langle B : b' \rangle
\]
Application 1: Factorized databases

- **Factorized databases**: implicit representation of database tables

\[
\begin{array}{cc}
A & B \\
a & b \\
a & b' \\
\end{array}
\]

\[
\{(\langle A : a \rangle, \langle B : b \rangle), (\langle A : a \rangle, \langle B : b' \rangle)\}
\]

\[
\{\langle B : b \rangle, \langle B : b' \rangle\}
\]

\[
\times
\]

\[
\bigcup
\]

\[
\langle A : a \rangle \\
\langle B : b \rangle \\
\langle B : b' \rangle
\]

\[
\text{represented by}
\]
Application 1: Factorized databases

- **Factorized databases**: implicit representation of database tables

\[
\begin{array}{ccc}
A & B \\
a & b & \left\{ \langle A : a \rangle, \langle B : b \rangle \rangle, \langle A : a \rangle, \langle B : b' \rangle \rangle \right\} \\
a & b' \\
\end{array}
\]

represented by

\[
\begin{array}{ccc}
A : a \\
\times \\
B : b \setminus \left\{ \langle B : b \rangle, \langle B : b' \rangle \rangle \right\} \\
B : b' \\
\end{array}
\]


- Relational product \( \times \)
- Relational union \( \cup \)

Theorem (Strenghtens result of [Olteanu and Závodn `y, /two.osf/zero.osf/one.osf/five.osf])

Given a deterministic factorized representation, we can enumerate its tuples with linear preprocessing and constant delay.
Application 1: Factorized databases

- Factorized databases: implicit representation of database tables

\[
\begin{align*}
\{\langle A : a \rangle, \langle B : b \rangle \}, \{\langle A : a \rangle, \langle B : b' \rangle \}\}
\end{align*}
\]

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

- Relational product \( \times \)
- Relational union \( \cup \)
- Deterministic: We do not obtain the same tuple multiple times

Theorem (Strenghtens result of [Olteanu and Závodn `y, /two.osf/zero.osf/one.osf/five.osf])

Given a deterministic factorized representation, we can enumerate its tuples with linear preprocessing and constant delay.
Application 1: Factorized databases

- **Factorized databases:** implicit representation of database tables

\[
\{((A : a), (B : b)), ((A : a), (B : b'))\}
\]

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

represented by \(\langle A : a \rangle\) \(\langle B : b \rangle\) \(\langle B : b' \rangle\)

- **Relational product** \(\times\)
- **Relational union** \(\cup\)
- **Deterministic:** We do not obtain the same tuple multiple times

**Theorem (Strengthen 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

• Compute the results \((a, b, c)\) of a query \(Q(x, y, z)\) on a database \(D\)
Application 2: Query evaluation

• Compute the results \((a, b, c)\) of a query \(Q(x, y, z)\) on a database \(D\)

• Assumption: the database has bounded treewidth
  → Captures trees, words, etc.
Application 2: Query evaluation

• Compute the results \((a, b, c)\) of a query \(Q(x, y, z)\) on a database \(D\)

• Assumption: the database has bounded treewidth
  → Captures trees, words, etc.

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

• Compute the results \((a, b, c)\) of a query \(Q(x, y, z)\) on a database \(D\)

• **Assumption:** the database has **bounded treewidth**
  → Captures trees, words, etc.

• 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 database \(D\)

- **Assumption:** the database has bounded treewidth
  - Captures trees, words, etc.

- 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

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

Given a MSO query \(Q\) and a database \(D\), the results of \(Q\) on \(D\) can be enumerated with linear preprocessing in \(D\) and linear delay in each answer (→ constant delay for free first-order variables)
Proof techniques
Proof overview

Preprocessing phase:

\[ \neg x \land z \]

Circuit

\[ \neg \land \]

\[ x \]

\[ z \]

\[ \lor \]

\[ \neg \land \]

\[ x \]

\[ z \]

\[ \lor \]

\[ \land \]

\[ x \]

\[ z \]

\[ \land \]

\[ x \]

\[ y \]

\[ z \]

v-tree

Translation (linear-time)

\[ \neg x \land z \]

Circuit in zero-suppressed semantics

Normalization (linear-time)

\[ \neg x \land z \]

Normalized circuit

Enumeration phase:

\[ \land \]

\[ x \]

\[ z \]

Normalized circuit

Enumeration (linear delay in each result)

A B C

a b c

a b' c

Results
Proof overview

Preprocessing phase:

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

\[ \text{Circuit} \]

Translation (linear-time)

\[ \lor \land x z \]

\[ \text{Circuit in zero-suppressed semantics} \]

Enumeration phase:

\[ \land x z \]

\[ \text{Normalized circuit} \]

\[ \lor \land x z \]

\[ \text{Enumeration (linear delay in each result)} \]

Results

\[ \text{/one.osf /three.osf} \]
Proof overview

Preprocessing phase:

Circuit

\[ \neg x \land z \]

∨

\[ \neg x \]

∧

\[ x \land z \]

v-tree

Translation (linear-time)

\[ \neg x \land z \]

Circuit in zero-suppressed semantics

\[ \neg x \]

∧

\[ x \land z \]

Normalization (linear-time)

\[ x \land z \]

Normalized circuit

\[ x \land z \]

\[ a \land b \land c \]

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

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

\[ a \land b \land c \]

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

\[ a \land b \land c \]

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

\[ a \land b \land c \]

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

\[ a \land b \land c \]

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

\[ a \land b \land c \]

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

\[ a \land b \land c \]

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

\[ a \land b \land c \]

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

\[ a \land b \land c \]

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

\[ a \land b \land c \]

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

\[ a \land b \land c \]

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

\[ a \land b \land c \]

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

\[ a \land b \land c \]

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

\[ a \land b \land c \]

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

\[ a \land b \land c \]

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

\[ a \land b \land c \]

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

\[ a \land b \land c \]

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

\[ a \land b \land c \]

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

\[ a \land b \land c \]

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

\[ a \land b \land c \]

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

\[ a \land b \land c \]

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

\[ a \land b \land c \]

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

\[ a \land b \land c \]
Proof overview

Preprocessing phase:
- Circuit
- v-tree
- Translation (linear-time)
- Circuit in zero-suppressed semantics
- Normalization (linear-time)
- Normalized circuit

Enumeration phase:
- Normalized circuit
Proof overview

Preprocessing phase:

- Circuit
- v-tree

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

Enumeration phase:

- Normalized circuit

Enumeration (linear delay in each result) → Results

Translation (linear-time)

\( \vee \neg x \land z \)

Circuit

\( \neg x \land z \)

v-tree

\( \land x \land z \)

\( \land x \land z \)

\( x \land z \)

\( x \land z \)

\( a \quad b \quad c \)

\( a \quad b' \quad c \)
Zero-suppressed semantics

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

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

Zero-suppressed semantics
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 $\cup$

Diagram:

$\land$

$\lor$

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

\[\{\{x, y\}, \{x, z\}\}\]

Special zero-suppressed semantics for circuits:

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

\[
\begin{align*}
\land & \{\{x, y\}, \{x, z\}\} \\
\lor & \{\{y\}, \{z\}\}
\end{align*}
\]
Zero-suppressed semantics

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 $\cup$

- \textbf{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

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

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 $\{x, z\}$
Enumerating assignments in the zero-suppressed semantics

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 $\textcircled{x}$ :
Enumerating assignments in the zero-suppressed semantics

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 $x$ : enumerate $\{x\}$ and stop

OR-gate

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 $g$

→ 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**

$g \lor g'$

**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

\[
g \lor g'
\]

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

Determinism: no duplicates

AND-gate

\[
g \land g'
\]

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

OR-gate $g \lor g'$

AND-gate $g \land g'$

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

Decomposability: no duplicates
Conclusion
Summary and conclusion

Summary:

- **Usual approach**: develop enumeration algorithms by hand
Summary and conclusion

Summary:

- **Usual approach:** develop enumeration algorithms by hand
- **Proposed approach:**

Future work:

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

Thanks for your attention!
Summary and conclusion

Summary:

• **Usual approach:** develop enumeration algorithms by hand
• **Proposed approach:**
  • Develop linear-time compilation algorithm to *circuits*

Future work:

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

Thanks for your attention!
Summary and conclusion

Summary:

- **Usual approach:** develop enumeration algorithms by hand
- **Proposed approach:**
  - Develop linear-time compilation algorithm to **circuits**
  - Use **restricted** circuit classes (structured d-DNNF)

Future work:

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

Thanks for your attention!
Summary and conclusion

Summary:

- **Usual approach:** develop enumeration algorithms by hand
- **Proposed approach:**
  - Develop linear-time compilation algorithm to **circuits**
  - Use **restricted** circuit classes (structured d-DNNF)
  - Develop general enumeration results on circuits

Future work:

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

Thanks for your attention!
Summary and conclusion

Summary:

• **Usual approach:** develop enumeration algorithms by hand

• **Proposed approach:**
  • Develop linear-time compilation algorithm to **circuits**
  • Use **restricted** circuit classes (structured d-DNNF)
  • Develop general enumeration results on circuits

Future work:

• **Theory:** handle **updates** on the structure

• **Practice:** implement the technique with automata
Summary and conclusion

Summary:

• **Usual approach:** develop enumeration algorithms by hand

• **Proposed approach:**
  • Develop linear-time compilation algorithm to **circuits**
  • Use **restricted** circuit classes (structured d-DNNF)
  • Develop general enumeration results on circuits

Future work:

• **Theory:** handle **updates** on the structure

• **Practice:** implement the technique with automata

Thanks for your attention!
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 $S(g) = \emptyset$
Normalization: handling $\emptyset$

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

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

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.

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

→ 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

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

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 \( \{\} \) 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 / one.osf

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

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

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

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 \( \{\} \) we waste time in chains of AND-gates

- **Solution:**
  - split \( g \) between \( S(g) \cap \{\} \) and \( S(g) \setminus \{\} \) (homogenization)
Normalizing: 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 $\emptyset$ 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 1

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 1

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 1

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

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**

![Diagram of OR-hierarchies](image_url)
Normalization: handling OR-hierarchies

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

Solution: compute reachability index
- **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

\[
\begin{align*}
  & x \\
  & \quad \lor \\
  & \quad \quad y \\
  & \quad \lor \\
  & \quad \quad z
\end{align*}
\]

- 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 \textit{v-tree}
- Add explicitly \textit{untested variables}

\[
\neg x \lor y \lor \neg z
\]

Problem: quadratic blowup
Solution:
- Order \textless on variables in the \textit{v-tree} (\(x \textless y \textless z\))
- Interval \([x, z]\)
- Range gates to denote \(\bigvee [x, z]\) in constant space
• This is where we use the \textit{v-tree}
• Add explicitly \textit{untested variables}
Translating to zero-suppressed semantics

- This is where we use the \textit{v-tree}
- Add explicitly \textit{untested variables}

\[ x \lor y \lor z \]

- \textbf{Problem: quadratic blowup}

\[ x \land y \land z \]

\[ x \lor y \lor z \]
Translating to zero-suppressed semantics

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

**Problem:** quadratic blowup

**Solution:**
- **Order** < on variables in the v-tree
  - \((x < y < z)\)
- **Interval** \([x, z]\)
- **Range gates** to denote \(\bigvee[x, z]\) in constant space
Bagan, G. (2006). **MSO queries on tree decomposable structures are computable with linear delay.**
In *CSL*.
