NP-Preprocessing

Pierre Marquis

CRIL, U. Artois & CNRS,
Institut Universitaire de France
France

Overview

Introduction

Reducing CNF Formulae

P-Preprocessings

NP-Preprocessings

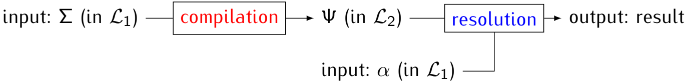
Combining Preprocessings

Implicit GDR thanks to Definability

Knowledge Compilation vs. Preprocessing

Two **preprocessing** approaches for circumventing the complexity of computationally hard tasks

- **knowledge compilation**



Knowledge Compilation vs. Preprocessing

Two **preprocessing** approaches for circumventing the complexity of computationally hard tasks

- **knowledge compilation**

  input: Σ (in $\mathcal{L}_1$) → $\boxed{\text{compilation}}$ → Ψ (in $\mathcal{L}_2$) → $\boxed{\text{resolution}}$ → output: result

  input: α (in $\mathcal{L}_1$) → $\boxed{\text{resolution}}$

- **preprocessing**

  input: Σ, α (in $\mathcal{L}_1$) → $\boxed{\text{preprocessing}}$ → Ψ (in $\mathcal{L}_1$) → $\boxed{\text{resolution}}$ → output: result

Knowledge Compilation vs. Preprocessing

Two **preprocessing** approaches for circumventing the complexity of computationally hard tasks

- **knowledge compilation**

  input: Σ (in $\mathcal{L}_1$) — compilation → Ψ (in $\mathcal{L}_2$) — resolution → output: result

  input: α (in $\mathcal{L}_1$) — resolution

- **preprocessing**

  input: Σ, α (in $\mathcal{L}_1$) — preprocessing → Ψ (in $\mathcal{L}_1$) — resolution → output: result

- The two approaches can be **combined**

Knowledge Compilation vs. Preprocessing

- Main resemblances:
  - making the resolution of the instance computationally easier once the preprocessing step has been achieved
  - no guarantee of success

Knowledge Compilation vs. Preprocessing

- Main resemblances:
  - making the resolution of the instance computationally easier once the preprocessing step has been achieved
  - no guarantee of success

- Main differences:
  - "hard" part vs. "easy" part of the solving process
  - handling of the variable part α of the input (does not exist in general, can be preprocessed as well or not)

Computationally Hard Tasks = ?

- SAT
  - Input: a CNF formula Σ
  - Output: 1 if Σ is satisfiable, 0 otherwise
  - The canonical NP-complete problem!
- #SAT
  - Input: a CNF formula Σ (plus eventually a satisfiable term α)
  - Output: the number of models of Σ (conditioned by α)
  - The canonical #P-complete problem!

Satisfiability

- $\Sigma \mapsto 1$ if Σ has a model, 0 otherwise

Satisfiability

- $\Sigma \mapsto 1$ if Σ has a model, 0 otherwise
- $\Sigma = (x \vee y) \wedge (\neg y \vee z)$

Satisfiability

- $\Sigma \mapsto 1$ if Σ has a model, 0 otherwise
- $\Sigma = (x \vee y) \wedge (\neg y \vee z)$
- Σ is satisfiable since (for instance) 011 is a model of Σ

Model Counting

- $\Sigma \mapsto \|\Sigma\| = ?$

Model Counting

- $\Sigma \mapsto \|\Sigma\| = ?$
- $\Sigma = (x \vee y) \wedge (\neg y \vee z)$

Model Counting

- $\Sigma \mapsto \|\Sigma\| = ?$
- $\Sigma = (x \vee y) \wedge (\neg y \vee z)$
- The models of Σ over $\{x, y, z\}$ are :

  011
  100
  101
  111

Model Counting

- $\Sigma \mapsto \|\Sigma\| \; = ?$
- $\Sigma = (x \vee y) \wedge (\neg y \vee z)$
- The models of Σ over $\{x, y, z\}$ are :

  011
  100
  101
  111

- $\|\Sigma\| = 4$

Model Counting

- Counting the models of a propositional formula is a key task for a number of problems (especially in AI):
  - probabilistic inference
  - stochastic planning
  - ...

Model Counting

- Counting the models of a propositional formula is a key task for a number of problems (especially in AI):
  - probabilistic inference
  - stochastic planning
  - ...
- However #SAT is a computationally hard task: #P-complete

Model Counting

- Counting the models of a propositional formula is a key task for a number of problems (especially in AI):
  - probabilistic inference
  - stochastic planning
  - ...
- However #SAT is a computationally hard task: #P-complete
- Even for subsets of formulae for which SAT is easy (e.g., monotone Krom formulae)!

Model Counting

- Counting the models of a propositional formula is a key task for a number of problems (especially in AI):
  - probabilistic inference
  - stochastic planning
  - ...
- However #SAT is a computationally hard task: #P-complete
- Even for subsets of formulae for which SAT is easy (e.g., monotone Krom formulae)!
- The "power" of counting and its complexity are reflected by **Toda's theorem:**

Seinosuke Toda (Gödel Prize 1998):

$$\mathsf{PH} \subseteq \mathsf{P}^{\#\mathsf{P}}$$

Model Counting

- **Many model counters** have been developed:
  - Exact model counters:
    - search-based: `Cachet`, `SharpSAT`, etc.,
    - compilation-based: `C2D`, `Dsharp`, `D4`, etc.
    - ...
  - Approximate model counters (`SampleCount`, etc.)
  - ...

Knowledge Compilation vs. Preprocessing for Model Counting

- **knowledge compilation**

input: Σ (in CNF) — $\boxed{\text{compilation}}$ → Ψ (in d-DNNF) — $\boxed{\text{model counting}}$ → $\|\Sigma \mid \alpha\|$

input: α (a consistent term)

Knowledge Compilation vs. Preprocessing for Model Counting

- **knowledge compilation**

  input: Σ (in CNF) — $\boxed{\text{compilation}}$ $\rightarrow$ Ψ (in d-DNNF) — $\boxed{\text{model counting}}$ $\rightarrow$ $\|\Sigma \mid \alpha\|$

  input: α (a consistent term)

- **preprocessing**

  input: Σ, α (in CNF) — $\boxed{p}$ $\rightarrow$ Ψ (in CNF) — $\boxed{\text{model counting}}$ $\rightarrow$ $\|\Sigma \mid \alpha\|$

P-Preprocessings

- Polynomial-time preprocessings
- Can prove useful for SAT and #SAT

- Related to the notion of kernelization: given a parameterized problem $(L \subseteq V^*, \kappa : V^* \to \mathbb{N})$, a kernelization for L is a polynomial-time algorithm p that takes an instance $\Sigma \subseteq V^*$ with parameter $\kappa(\Sigma)$, and maps it to an instance $p(\Sigma) \subseteq V^*$ such that $\Sigma \in L$ if and only if $p(\Sigma) \in L$ and the size of $p(\Sigma)$ is upper bounded by $f(\kappa(\Sigma))$ for some computable function f
- If L is decidable, then L is fixed-parameter tractable for parameter $\kappa(.)$ if and only if L has a kernelization

Dozens of P-Preprocessings

- Vivification (VI) and a light form of it, called Occurrence Elimination (OE),
- Gate Detection and Replacement (GDR)
- Pure Literal Elimination (PLE)
- Variable Elimination (VE)
- Blocked Clause Elimination (BCE)
- Covered Clause Elimination (CCE)
- Failed Literal Elimination (FLE)
- Self-Subsuming Resolution (SSR)
- Hidden Literal Elimination (HLE)
- Subsumption Elimination (SE)
- Hidden Subsumption Elimination (HSE)
- Asymmetric Subsumption Elimination (ASE)
- Tautology Elimination (TE)
- Hidden Tautology Elimination (HTE)
- Asymmetric Tautology Elimination (ATE)
- ...

Use in State-of-the-Art SAT Solvers

- `Glucose` (exploits the `SatELite` preprocessor)
- `Lingeling` (has an internal preprocessor)
- `Riss` (use of the `Coprocessor` preprocessor)
- ...

Overview

The CNF Input Assumption

SAT solvers and model counters typically considers CNF inputs

- The CNF assumption is not restrictive
- Every circuit Ψ can be turned into a CNF formula Σ in linear time
- The translation requires the introduction of new variables $Y = \{y_1, \ldots, \}$ and preserves
  - the queries over the alphabet of the input circuit (Plaisted/Greenbaum)

$$\Psi \equiv \exists Y.\Sigma$$

  - and the number of models of the input (Tseitin)

Translation into CNF: Tseitin

$$\Psi = (x_1 \wedge \bar{x}_2) \vee (x_2 \wedge x_3)$$
$$T(\Psi) \equiv (y_1 \vee y_2) \wedge (y_1 \Leftrightarrow (x_1 \wedge \bar{x}_2)) \wedge (y_2 \Leftrightarrow (x_2 \wedge x_3))$$

$T(\Psi) =$

$y_1 \vee y_2$

$\bar{y}_1 \vee x_1$

$\bar{y}_1 \vee \bar{x}_2$

$y_1 \quad \vee \bar{x}_1 \vee x_2$

$\bar{y}_2 \vee x_2$

$\bar{y}_2 \vee x_3$

$y_2 \quad \vee \bar{x}_2 \vee \bar{x}_3$

Translation into CNF: Plaisted/Greenbaum

$\Psi = (x_1 \wedge \bar{x}_2) \vee (x_2 \wedge x_3)$
$PG(\Psi) \equiv (y_1 \vee y_2) \wedge (y_1 \Rightarrow (x_1 \wedge \bar{x}_2)) \wedge (y_2 \Rightarrow (x_2 \wedge x_3))$

$PG(\Psi) =$

$y_1 \vee y_2$
$\bar{y}_1 \vee x_1$
$\bar{y}_1 \vee \bar{x}_2$
$\bar{y}_2 \vee x_2$
$\bar{y}_2 \vee x_3$

Reducing What?

$\texttt{CNF}\ \Sigma \mapsto \texttt{CNF}\ p(\Sigma)$

- What are the connections between Σ and $p(\Sigma)$?
- Removing clauses from Σ
- Removing literals in the clauses of Σ
- Updating Σ in another way
- ...

Looking for IES or Minimal CNF is often too Expensive

- A clause δ of a CNF Σ is redundant if and only if $\Sigma \setminus \{\delta\} \models \delta$
- A CNF Σ is irredundant if and only if it does not contain any redundant clause
- A subset Σ' of a CNF Σ is an irredundant equivalent subset (IES) of Σ if and only if Σ' is irredundant and $\Sigma' \equiv \Sigma$

- Deciding whether a CNF Σ is irredundant is NP-complete
- Deciding whether a CNF Σ' is an irredundant equivalent subset (IES) of a CNF Σ is D^p-complete
- Given an integer k, deciding whether a CNF Σ has an IES of size at most k is Σ_2^p-complete
- Given an integer k, deciding whether there exists a CNF formula Σ' with at most k literals (or with at most k clauses) equivalent to a given CNF Σ is Σ_2^p-complete

Redundancy can be Useful!

Redundancy can be helpful (favoring the unit propagation power by adding empowering clauses)

- $x_3 \vee x_4$ is a logical consequence of
  $\Sigma =$
  $x_1 \vee x_2 \vee x_3$
  $\bar{x}_1 \vee x_2 \vee x_4$
  $x_1 \vee \bar{x}_2 \vee x_3$
  $\bar{x}_1 \vee \bar{x}_2 \vee x_4$
- Assuming $\bar{x}_3 \wedge \bar{x}_4$, there is no unit refutation from Σ
- If $x_3 \vee x_4$ (or $x_2 \vee x_3 \vee x_4$ or $\bar{x}_2 \vee x_3 \vee x_4$) is added to Σ, a unit refutation from Σ under the assumption $\bar{x}_3 \wedge \bar{x}_4$ exists

- Learning clauses is a key ingredient for efficient SAT solvers based on the CDCL architecture!

Preserving What?

- Logical equivalence
- Number of models
- Satisfiability
- ...

Levels of Preservation

$A \rightarrow B$: if a preprocessing p preserves A, then p preserves B

$$\begin{array}{c}\text{Logical equivalence}\\ \downarrow \\ \text{Number of models}\\ \downarrow \\ \text{Satisfiability}\end{array}$$

Properties of Preprocessings

- Level of preservation (logical equivalence, number of models, satisfiability)
- Confluence: is $p(\Sigma)$ sensitive w.r.t. the way clauses and literals in them are listed in Σ?
- Projectiveness: do we have $p(p(\Sigma)) = p(\Sigma)$?
  (important to decide whether iterating p makes sense or not)

Measuring the Impact of a Preprocessing

Several measures for the reduction achieved can be considered:

- The number of variables in the input `CNF` Σ
- The size of Σ (the number of literals or the number of clauses in it)
- The treewidth of the primal graph of Σ
- The value of other structural parameters for Σ
- ...

Example: Subsumption Elimination

A clause δ_1 subsumes a clause δ_2
if every literal of δ_1 is a literal of δ_2

$SE : (x_1 \vee x_2) \wedge (x_1 \vee x_2 \vee \bar{x}_3) \mapsto x_1 \vee x_2$

- Preserves logical equivalence
- Hence preserves the number of models of the input (over the original alphabet), and its satisfiability
- Is confluent and projective

- $\#var(\Sigma) \geq \#var(\mathtt{SE}(\Sigma))$
- $\#lit(\Sigma) \geq \#lit(\mathtt{SE}(\Sigma))$
- $tw(\Sigma) \geq tw(\mathtt{SE}(\Sigma))$

Example: Pure Literal Elimination

A literal l is pure in Σ if every occurrence of the corresponding variable has the same polarity

$PLE : (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_2 \vee \bar{x}_3) \mapsto \bar{x}_2 \vee \bar{x}_3$

- Preserves the satisfiability of the input
- Does not preserve its number of models or logical equivalence
- Is confluent and projective

- $\#var(\Sigma) \geq \#var(\mathtt{PLE}(\Sigma))$
- $\#lit(\Sigma) \geq \#lit(\mathtt{PLE}(\Sigma))$
- $tw(\Sigma) \geq tw(\mathtt{PLE}(\Sigma))$

Estimating the Amount of Reduction Achieved

- For each p, the impact of p is evaluated empirically (and quantitatively) on 182 `CNF` instances from the SAT LIBrary, gathered into 9 data sets, as follows:
  - Bayesian networks (60)
  - BMC (11) (Bounded Model Checking)
  - Circuit (28)
  - Configuration (12)
  - Handmade (28)
  - Planning (17)
  - Random (13)
  - Scheduling (6)
  - Qif (7) (Quantitative Information Flow analysis - security)
- Cluster of Intel Xeon E5-2643 (3.30 GHz) processors with 32 GiB RAM on Linux CentOS
- Time-out =1h
- Memory-out = 7.6 GiB

Overview

Overview

Occurrence Reduction (OR)

- Occurrence reduction aims to remove some literals in the clauses of Σ
- In order to determine whether a literal ℓ_{j+1} can be removed from a clause α of Σ, the approach consists in determining whether the clause which coincides with α except that ℓ_{j+1} has been replaced by $\sim\ell_{j+1}$ is a logical consequence of Σ
- This is done by determining whether $\Sigma \wedge \ell_{j+1} \wedge \sim\ell_1 \wedge \ldots \wedge \sim\ell_j$ is contradictory
- When this is the case, ℓ_{j+1} can be removed from α without questioning logical equivalence
- bcp is used as an incomplete yet efficient method to solve the entailment problem

Algorithm 1: OR Occurrence reduction

input : a CNF formula Σ

output: a CNF formula equivalent to Σ

$\mathcal{L} \leftarrow \mathit{sort}(\mathit{Lit}(\Sigma))$;

foreach $\ell \in \mathcal{L}$ **do**

  foreach $\alpha \in \Sigma$ *s.t.* $\ell \in \alpha$ **do**

    if $\emptyset \in \mathtt{bcp}(\Sigma \cup \{\ell\} \cup \{\sim(\alpha \setminus \{\ell\})\})$ **then**

      $\Sigma \leftarrow (\Sigma \setminus \{\alpha\}) \cup \{\alpha \setminus \{\ell\}\}$;

return Σ

Occurrence Reduction (OR): Example

$$\Sigma = \begin{array}{ll} a \vee f & a \vee b \vee c \\ b \vee d \vee e & c \vee \neg d \vee e \\ b \vee d \vee \neg e & c \vee \neg d \vee \neg e \end{array}$$

$$\mathcal{L} = (b, c, e, \neg e, d, \neg d, a, f, \neg a, \neg b, \neg c)$$

$$\mathtt{OR}(\Sigma) = \begin{array}{ll} a \vee f & b \vee c \\ b \vee d & c \vee \neg d \\ b \vee d & c \vee \neg d \end{array}$$

$a \vee b \vee c$ is reduced to $b \vee c$ and every occurrence of e has been removed

Properties of OR

- Preserves logical equivalence
- Neither is confluent nor is projective
- $\#var(\Sigma) \geq \#var(\texttt{OR}(\Sigma))$
- $\#lit(\Sigma) \geq \#lit(\texttt{OR}(\Sigma))$
- $tw(\Sigma) \geq tw(\texttt{OR}(\Sigma))$

OR: Reduction of the Number of Variables

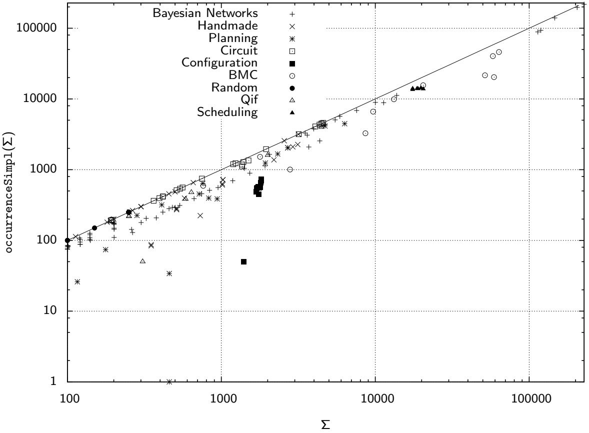


FIGURE: Comparing $\#var(\Sigma)$ with $\#var(\texttt{OR}(\Sigma))$.

OR: Reduction of the Size

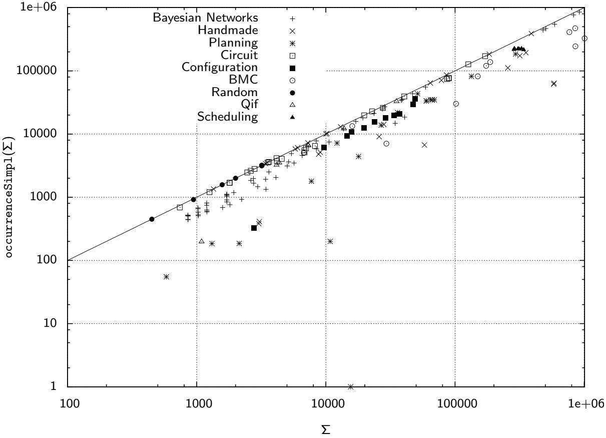


FIGURE: Comparing $\#lit(\Sigma)$ with $\#lit(\mathtt{OR}(\Sigma))$.

OR: Reduction of the Treewidth

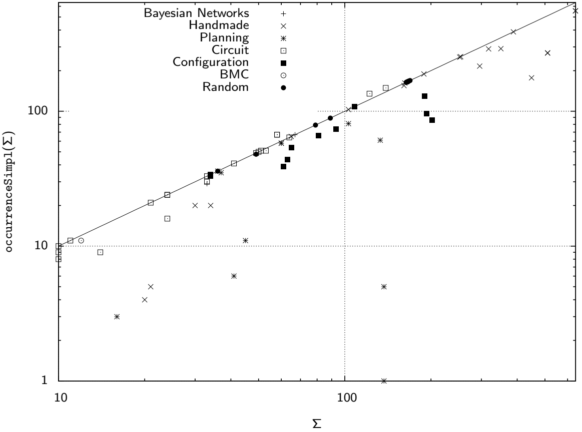


Figure: Comparing $tw(\Sigma)$ with $tw(\mathtt{OR}(\Sigma))$.

Overview

Vivification (VI)

- Vivification aims to reduce Σ, i.e., to remove some clauses and some literals in Σ while preserving equivalence
- Given a clause $\alpha = \ell_1 \vee \ldots \vee \ell_k$ of Σ two rules are used in order to determine whether α can be removed from Σ or simply shortened
- Let α' be any subclause of α
- On the one hand, if for any $j \in 0, \ldots, k-1$, one can prove using bcp that $\Sigma \setminus \{\alpha\} \models \alpha'$, then for sure α is entailed by $\Sigma \setminus \{\alpha\}$ so that α can be removed from Σ
- On the other hand, if one can prove using bcp that $\Sigma \setminus \{\alpha\} \models \alpha' \vee \sim\ell_{j+1}$, then ℓ_{j+1} can be removed from α without questioning equivalence

Vivification (VI)

Algorithm 2: `VI`

input : a `CNF` formula Σ

output: a `CNF` formula equivalent to Σ

foreach $\alpha \in \Sigma$ **do**
  $\Sigma \leftarrow \Sigma \setminus \{\alpha\}$;
  $\alpha' \leftarrow \bot$;
  $\mathcal{I} \leftarrow \mathtt{bcp}(\Sigma)$;
  while $\exists \ell \in \alpha$ *s.t.* $\sim\ell \notin \mathcal{I}$ *and* $\alpha' \neq \top$ **do**
    $\alpha' \leftarrow \alpha' \vee \ell$;
    $\mathcal{I} \leftarrow \mathtt{bcp}(\Sigma \cup \{\sim\alpha'\})$;
    if $\emptyset \in \mathcal{I}$ **then** $\alpha' \leftarrow \top$;
  $\Sigma \leftarrow \Sigma \cup \{\alpha'\}$;
return Σ

Vivification (VI): Example

$\Sigma =$

$a \vee b \vee c \vee d$

$a \vee b \vee c$

$a \vee \neg d$

Assume that the variables are processed w.r.t. the ordering $d < c < b < a$

$\mathtt{VI}(\Sigma) =$

$a \vee b \vee c$

$a \vee \neg d$

The effect of VI on Σ is to eliminate the first clause $a \vee b \vee c \vee d$

Properties of VI

- Preserves logical equivalence
- Neither is confluent nor is projective
- $\#var(\Sigma) \geq \#var(\mathtt{VI}(\Sigma))$
- $\#lit(\Sigma) \geq \#lit(\mathtt{VI}(\Sigma))$
- $tw(\Sigma) \geq tw(\mathtt{VI}(\Sigma))$

VI: Reduction of the Number of Variables

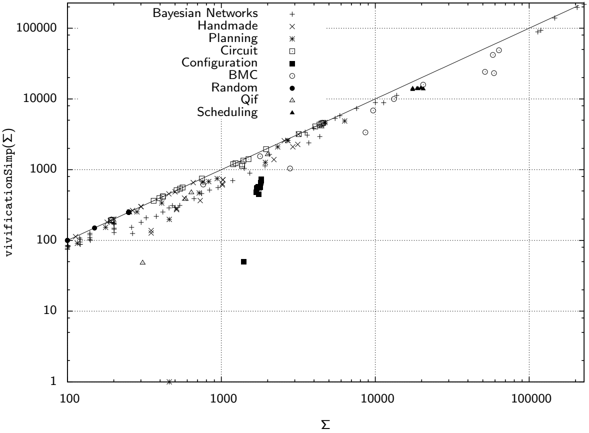


FIGURE: Comparing $\#var(\Sigma)$ with $\#var(\mathtt{VI}(\Sigma))$.

VI: Reduction of the Size

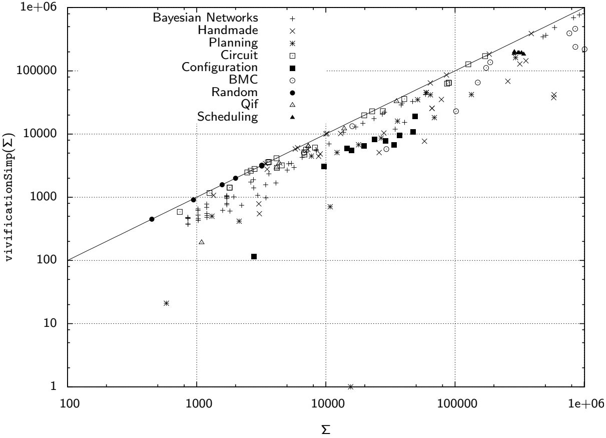


FIGURE: Comparing $\#lit(\Sigma)$ with $\#lit(\mathtt{VI}(\Sigma))$.

VI: Reduction of the Treewidth

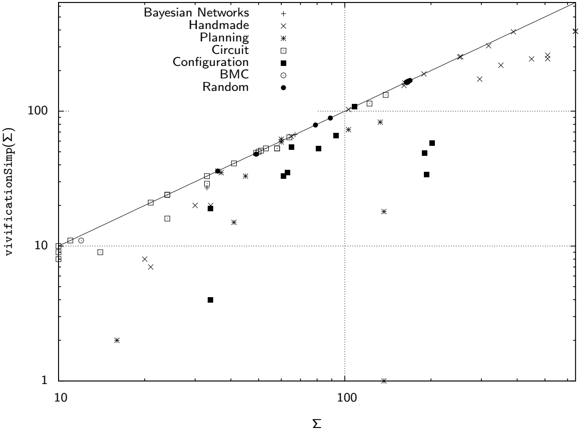


FIGURE: Comparing $tw(\Sigma)$ with $tw(\texttt{VI}(\Sigma))$.

Overview

Gate Detection and Replacement

A gate of Σ is a circuit $\ell \Leftrightarrow \beta$ such that $\Sigma \models \ell \Leftrightarrow \beta$
Σ and $\Sigma[\ell \leftarrow \beta]$ have the same number of models (but are not logically equivalent in general)

$$\Sigma = \begin{array}{l} \overline{x} \lor u \lor v \\ \overline{x} \lor \overline{y} \lor u \\ \overline{x} \lor \overline{z} \lor u \\ x \lor \overline{u} \\ y \lor z \lor \overline{u} \end{array}$$

Gate Detection and Replacement

A gate of Σ is a circuit $\ell \Leftrightarrow \beta$ such that $\Sigma \models \ell \Leftrightarrow \beta$
Σ and $\Sigma[\ell \leftarrow \beta]$ have the same number of models (but are not logically equivalent in general)

$$\Sigma = \begin{array}{l} \overline{x} \vee u \vee v \\ \overline{x} \vee \overline{y} \vee u \\ \overline{x} \vee \overline{z} \vee u \\ x \vee \overline{u} \\ y \vee z \vee \overline{u} \end{array} \qquad\qquad u \leftrightarrow (x \wedge (y \vee z))$$

Gate Detection and Replacement

A gate of Σ is a circuit $\ell \Leftrightarrow \beta$ such that $\Sigma \models \ell \Leftrightarrow \beta$
Σ and $\Sigma[\ell \leftarrow \beta]$ have the same number of models (but are not logically equivalent in general)

$$\Sigma = \quad \begin{array}{l} \overline{x} \lor u \lor v \\ \overline{x} \lor \overline{y} \lor u \\ \overline{x} \lor \overline{z} \lor u \\ x \lor \overline{u} \\ y \lor z \lor \overline{u} \end{array} \qquad\qquad u \leftrightarrow (x \land (y \lor z))$$

$$\Sigma \equiv (\overline{x} \lor u \lor v) \land (u \leftrightarrow (x \land (y \lor z)))$$

detection

Gate Detection and Replacement

A gate of Σ is a circuit $\ell \Leftrightarrow \beta$ such that $\Sigma \models \ell \Leftrightarrow \beta$
Σ and $\Sigma[\ell \leftarrow \beta]$ have the same number of models (but are not logically equivalent in general)

$$\Sigma = \begin{array}{l} \overline{x} \vee u \vee v \\ \overline{x} \vee \overline{y} \vee u \\ \overline{x} \vee \overline{z} \vee u \\ x \vee \overline{u} \\ y \vee z \vee \overline{u} \end{array} \qquad u \leftrightarrow (x \wedge (y \vee z))$$

$$\Sigma \equiv$$

$(\overline{x} \vee u \vee v) \wedge (u \leftrightarrow (x \wedge (y \vee z)))$	detection
$(\overline{x} \vee (x \wedge (y \vee z)) \vee v) \wedge (u \leftrightarrow (x \wedge (y \vee z)))$	replacement

Gate Detection and Replacement

A gate of Σ is a circuit $\ell \Leftrightarrow \beta$ such that $\Sigma \models \ell \Leftrightarrow \beta$

Σ and $\Sigma[\ell \leftarrow \beta]$ have the same number of models (but are not logically equivalent in general)

$$\Sigma = \begin{array}{l} \overline{x} \lor u \lor v \\ \overline{x} \lor \overline{y} \lor u \\ \overline{x} \lor \overline{z} \lor u \\ x \lor \overline{u} \\ y \lor z \lor \overline{u} \end{array} \qquad u \leftrightarrow (x \land (y \lor z))$$

$$\Sigma \equiv$$

$(\overline{x} \lor u \lor v) \land (u \leftrightarrow (x \land (y \lor z)))$	detection
$(\overline{x} \lor (x \land (y \lor z)) \lor v) \land (u \leftrightarrow (x \land (y \lor z)))$	replacement
$(\overline{x} \lor y \lor z \lor v) \land (u \leftrightarrow (x \land (y \lor z)))$	normalization

Gate Detection and Replacement

A gate of Σ is a circuit $\ell \Leftrightarrow \beta$ such that $\Sigma \models \ell \Leftrightarrow \beta$
Σ and $\Sigma[\ell \leftarrow \beta]$ have the same number of models (but are not logically equivalent in general)

$$\Sigma = \begin{array}{l} \overline{x} \vee u \vee v \\ \overline{x} \vee \overline{y} \vee u \\ \overline{x} \vee \overline{z} \vee u \\ x \vee \overline{u} \\ y \vee z \vee \overline{u} \end{array} \qquad u \leftrightarrow (x \wedge (y \vee z))$$

$\Sigma \equiv$

$(\overline{x} \vee u \vee v) \wedge (u \leftrightarrow (x \wedge (y \vee z)))$	detection
$(\overline{x} \vee (x \wedge (y \vee z)) \vee v) \wedge (u \leftrightarrow (x \wedge (y \vee z)))$	replacement
$(\overline{x} \vee y \vee z \vee v) \wedge (u \leftrightarrow (x \wedge (y \vee z)))$	normalization

$$\|\Sigma\| = \|\Sigma[u \leftarrow (x \wedge (y \vee z))]\| = \|\overline{x} \vee y \vee z \vee v\| = 15$$

Gate Detection and Replacement

- Gate detection and replacement proves to be a valuable preprocessing
- Only specific gates are sought for (literal equivalence, AND/OR gates, XOR gates)
- The replacement $\Sigma[\ell \leftarrow \beta]$ requires to turn the resulting formula into `CNF`
- It is implemented only if it it does not lead to increase the size of the input (a "small" increase can also be accepted)
- `bcp` (instead of a "full" SAT solver) is used for efficiency reasons

Literal Equivalence (LE)

- Literal equivalence aims to detect equivalences between literals using `bcp`
- For each literal ℓ, all the literals ℓ' which can be found equivalent to ℓ using `bcp` are replaced by ℓ in Σ
- Taking advantage of `bcp` makes it more efficient than a "syntactic detection" (if two binary clauses stating an equivalence between two literals ℓ and ℓ' occur in Σ, then those literals are found equivalent using `bcp`, but the converse does not hold)

Literal Equivalence (LE)

Algorithm 3: `LE`

input : a `CNF` formula Σ

output: a `CNF` formula Φ such that $\|\Phi\| = \|\Sigma\|$

$\Phi \leftarrow \Sigma$; Unmark all variables of Φ;

while $\exists \ell \in \mathit{Lit}(\Phi)$ *s.t.* $\mathit{var}(\ell)$ *is not marked* **do**

  `// detection`

  mark $\mathit{var}(\ell)$;

  $\mathcal{P}_\ell \leftarrow \mathtt{bcp}(\Phi \cup \{\ell\})$;

  $\mathcal{N}_\ell \leftarrow \mathtt{bcp}(\Phi \cup \{\sim\ell\})$;

  $\Gamma \leftarrow \{\ell \leftrightarrow \ell' | \ell' \neq \ell \text{ and } \ell' \in \mathcal{P}_\ell \text{ and } \sim\ell' \in \mathcal{N}_\ell\}$;

  `// replacement`

  foreach $\ell \leftrightarrow \ell' \in \Gamma$ **do**

    replace ℓ by ℓ' in Φ;

return Φ

Literal Equivalence (LE): Example

$\Sigma =$

$$\begin{array}{ll} a \vee b \vee c \vee \neg d & \neg a \vee \neg b \vee \neg c \vee d \\ a \vee b \vee \neg c & \neg a \vee \neg b \vee c \\ \neg a \vee b & a \vee \neg b \\ \neg e \vee \neg f \vee h & e \vee f \vee g \\ e \vee \neg g & \neg e \vee \neg h \end{array}$$

Assume that the variables of Σ are considered in the following ordering: $a < b < c < d < e < f < g < h$

The equivalences $(a \Leftrightarrow b) \wedge (b \Leftrightarrow c) \wedge (c \Leftrightarrow d) \wedge (e \Leftrightarrow \neg f)$ are detected

$\mathtt{LE}(\Sigma) =$

$$\begin{array}{ll} \neg f \vee \neg g & f \vee \neg h \end{array}$$

Properties of LE

- Preserves the number of models (but not logical equivalence)
- Neither is confluent nor is projective
- $\#var(\Sigma) \geq \#var(\mathtt{LE}(\Sigma))$
- $\#lit(\Sigma) \geq \#lit(\mathtt{LE}(\Sigma))$
- $tw(\Sigma) \not\geq tw(\mathtt{LE}(\Sigma))$

LE: Reduction of the Number of Variables

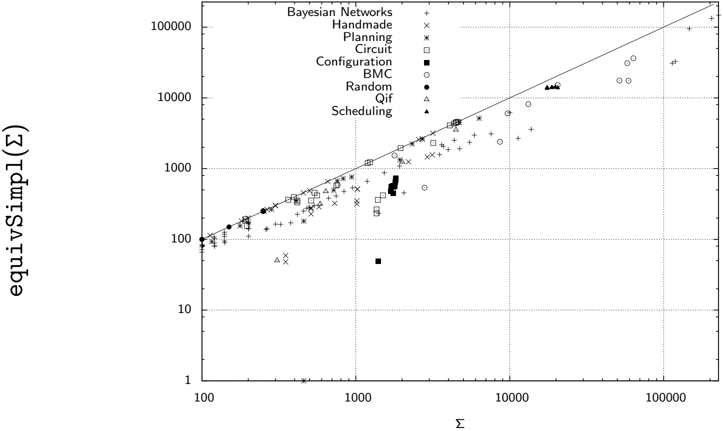


FIGURE: Comparing $\#var(\Sigma)$ with $\#var(\texttt{LE}(\Sigma))$.

LE: Reduction of the Size

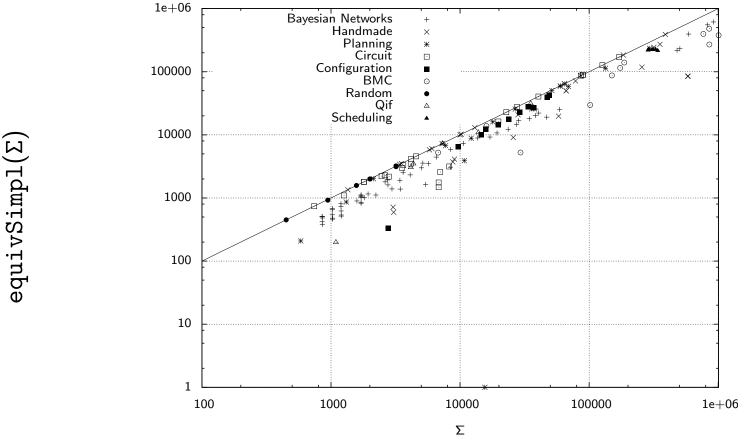


FIGURE: Comparing $\#lit(\Sigma)$ with $\#lit(\mathtt{LE}(\Sigma))$.

LE: Reduction of the Treewidth

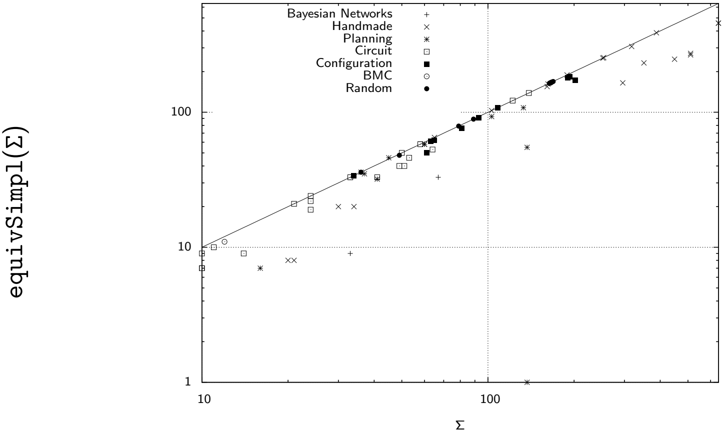


FIGURE: Comparing $tw(\Sigma)$ with $tw(\texttt{LE}(\Sigma))$.

AND/OR Gate Equivalence (AG)

- AND/OR gate equivalence aims to detect equivalences $\ell_i \Leftrightarrow \beta_i$ where β_i is a conjunction or a disjunction of literals
- `bcp` is used in this objective
- Unlike previous approaches based on pattern matching (i.e., when one looks for clauses encoding an AND gate or an OR gate), using `bcp` makes it more efficient (if clauses stating the presence of an AND/OR gate occur in Σ, then `AG` will find the gate – or a "subsuming" one – but the converse is not true)

Algorithm 4: AG

input : a CNF formula Σ
output : a CNF formula Φ such that $\|\Phi\| = \|\Sigma\|$

$\Phi \leftarrow \Sigma$;
// detection
$\Gamma \leftarrow \emptyset$; Unmark all literals of Φ;
while $\exists \ell \in \mathit{Lit}(\Phi)$ *s.t.* ℓ *is not marked* **do**
  mark ℓ;
  $\mathcal{P}_\ell \leftarrow (\texttt{bcp}(\Phi \cup \{\ell\}) \setminus (\texttt{bcp}(\Phi) \cup \{\ell\})) \cup \{\sim\ell\}$;
  if $\emptyset \in \texttt{bcp}(\Phi \cup \mathcal{P}_\ell)$ **then**
    let $\mathcal{C}_\ell \subseteq \mathcal{P}_\ell$ s.t. $\emptyset \in \texttt{bcp}(\Phi \cup \mathcal{C}_\ell)$ and $\sim\ell \in \mathcal{C}_\ell$;
    $\Gamma \leftarrow \Gamma \cup \{\ell \leftrightarrow \bigwedge_{\ell' \in \mathcal{C}_\ell \setminus \{\sim\ell\}} \ell'\}$;

// replacement
while $\exists \ell \leftrightarrow \beta \in \Gamma$ *st.* $|\beta| < \mathit{maxA}$ *and* $|\Phi[\ell \leftarrow \beta]| \leq |\Phi|$ **do**
  $\Phi \leftarrow \Phi[\ell \leftarrow \beta]$;
  $\Gamma \leftarrow \Gamma[\ell \leftarrow \beta]$;

return Φ

AND/OR Gate Equivalence (AG): Example

$\Sigma =$

$a \lor b \lor c \lor d$	$\neg a \lor \neg b$
$\neg a \lor b \lor \neg c$	$\neg a \lor b \lor c \lor \neg d$
$\neg a \lor e$	$a \lor f$

Assume that the literals are processed w.r.t. the literal ordering:
$a < \neg a < b < \neg b < c < \neg c < d < \neg d < e < \neg e < f < \neg f$

The gate $a \Leftrightarrow (\neg b \land \neg c \land \neg d)$ is detected and a is replaced by its definition

$\mathtt{AG}(\Sigma) =$

$b \lor c \lor d \lor e$	$\neg b \lor f$
$\neg c \lor f$	$\neg d \lor f$

Properties of AG

- Preserves the number of models (but not logical equivalence)
- Neither is confluent nor is projective
- $\#var(\Sigma) \geq \#var(\mathtt{AG}(\Sigma))$
- $\#lit(\Sigma) \geq \#lit(\mathtt{AG}(\Sigma))$
- $tw(\Sigma) \not\geq tw(\mathtt{AG}(\Sigma))$

AG: Reduction of the Number of Variables

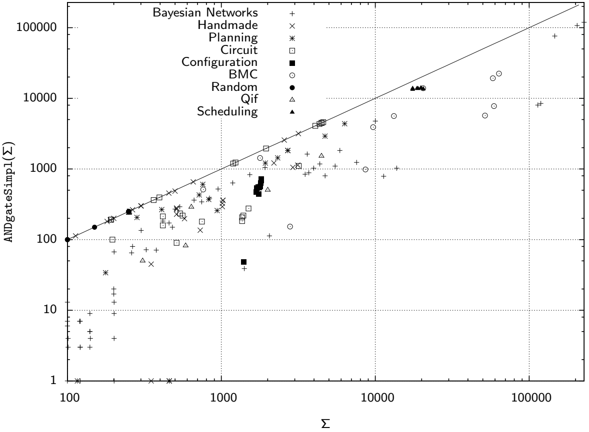


FIGURE: Comparing $\#var(\Sigma)$ with $\#var(\mathtt{AG}(\Sigma))$.

AG: Reduction of the Size

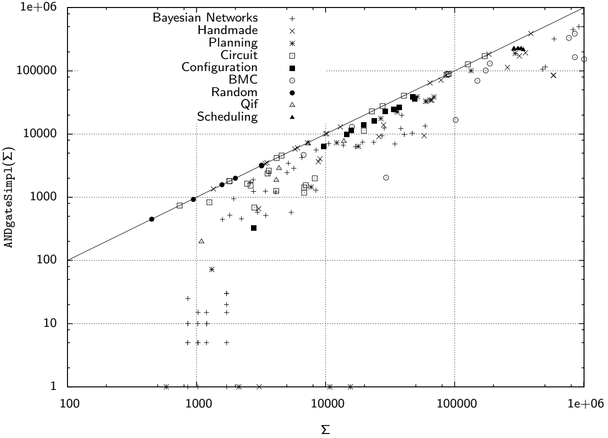


FIGURE: Comparing $\#lit(\Sigma)$ with $\#lit(\texttt{AG}(\Sigma))$.

AG: Reduction of the Treewidth

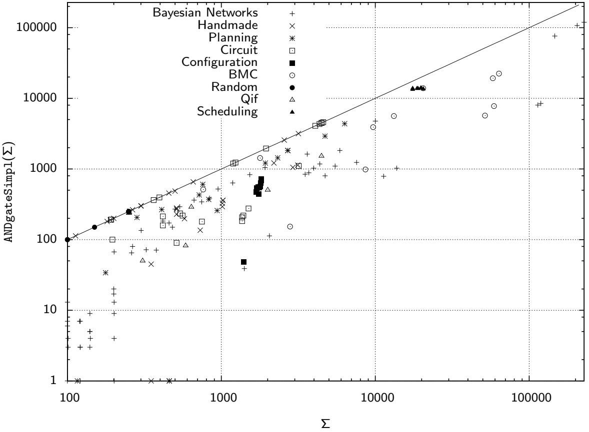


FIGURE: Comparing $tw(\Sigma)$ with $tw(\mathtt{AG}(\Sigma))$.

XOR Gate Equivalence (XG)

- XOR gate detection and replacement aims to detect equivalences $\ell_i \Leftrightarrow \chi_i$ where χ_i is a XOR disjunction of literals
- The detection is "syntactic" (XOR disjunctions of literals containing a limited number of literals are targeted)
- The resulting set of gates, which can be viewed as a set of XOR clauses since $\ell_i \leftrightarrow \chi_i$ is equivalent to $\sim\ell_i \oplus \chi_i$, is turned into reduced row echelon form using Gauss algorithm
- Every ℓ_i is replaced by its definition χ_i in Σ, provided that the normalization it involves does not generate "large" clauses

XOR Gate Equivalence (XG)

Algorithm 5: `XG`

input : a `CNF` formula Σ

output: a `CNF` formula Φ such that $\|\Phi\| = \|\Sigma\|$

$\Phi \leftarrow \Sigma$;

`// detection`

$\Gamma \leftarrow \{$XOR clauses syntactically detected$\}$;

`// Gaussian elimination`

$\Gamma \leftarrow \text{Gauss}(\{\ell_1 \leftrightarrow \chi_1, \ell_2 \leftrightarrow \chi_2, \dots, \ell_k \leftrightarrow \chi_k\})$

`// replacement`

for $i \leftarrow 1$ *to* k **do**

  if $\nexists \alpha \in \Phi[\ell_i \leftarrow \chi_i]$ *s.t.* $|\alpha| > maxX$ **then** $\Phi \leftarrow \Phi[\ell_i \leftarrow \chi_i]$;

return Φ

XOR Gate Equivalence (XG): Example

$$\Sigma = \begin{array}{ll} b \vee d & \neg b \vee \neg d \\ \neg a \vee \neg b \vee c & a \vee \neg b \vee \neg c \\ \neg a \vee b \vee \neg c & a \vee b \vee c \\ b \vee e & a \vee f \end{array}$$

Suppose that the two XOR gates $b \oplus d$ and $a \oplus b \oplus c$ are detected successively

$$\mathtt{XG}(\Sigma) = \begin{array}{ll} \neg d \vee e & c \vee d \vee f \\ \neg c \vee \neg d \vee f & \end{array}$$

The first six clauses which participate in the XOR gates are made valid through the replacement, the clause $b \vee e$ is replaced by $\neg d \vee e$, and the clause $a \vee f$ is replaced by the two clauses $c \vee d \vee f$ and $\neg c \vee \neg d \vee f$

Properties of XG

- Preserves the number of models (but not logical equivalence)
- Neither is confluent nor is projective
- $\#var(\Sigma) \geq \#var(\mathtt{XG}(\Sigma))$
- $\#lit(\Sigma) \geq \#lit(\mathtt{XG}(\Sigma))$
- $tw(\Sigma) \not\geq tw(\mathtt{XG}(\Sigma))$

XG: Reduction of the Number of Variables

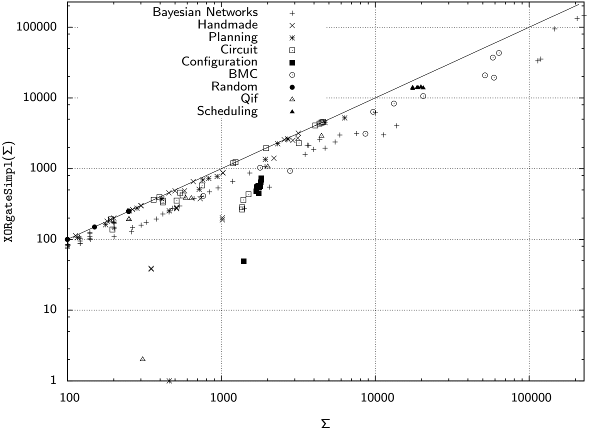


FIGURE: Comparing $\#var(\Sigma)$ with $\#var(\mathtt{XG}(\Sigma))$.

XG: Reduction of the Size

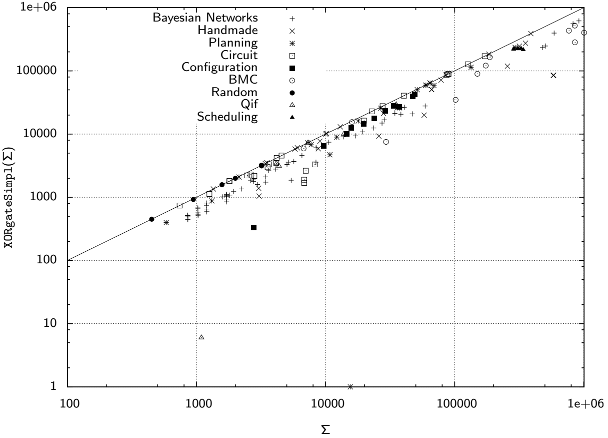


FIGURE: Comparing $\#lit(\Sigma)$ with $\#lit(\texttt{XG}(\Sigma))$.

XG: Reduction of the Treewidth

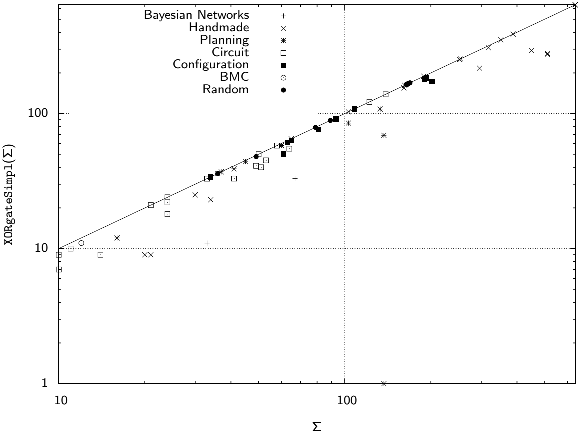


FIGURE: Comparing $tw(\Sigma)$ with $tw(\mathtt{XG}(\Sigma))$.

Overview

NP-Preprocessings

- Taking advantage of SAT solvers for simplifying the input CNF during a preprocessing step
- An option that makes sense when tackling problems which are seemingly located ”above NP” (like #SAT)

- Backbone identification
- Definability

Overview

Backbone Identification (BI)

- The backbone of a `CNF` formula Σ is the set of all literals which are implied by Σ when Σ is satisfiable, and is the empty set otherwise
- The purpose of the *BI* preprocessing is to make the backbone B of the input `CNF` formula Σ explicit, to conjoin it to Σ, and to use `bcp` (Boolean Constraint Propagation) on the resulting set of clauses

Backbone Identification (BI)

Algorithm 6: BI Backbone Identification

input : a CNF formula Σ

output: the CNF $\mathtt{bcp}(\Sigma \cup B)$, where $\mathcal{B}$ is the backbone of Σ

$\mathcal{B} \leftarrow \emptyset$;

$\mathcal{I} \leftarrow \mathtt{solve}(\Sigma)$;

while $\exists \ell \in \mathcal{I}$ *s.t.* $\ell \notin \mathcal{B}$ **do**

  $\mathcal{I}' \leftarrow \mathtt{solve}(\Sigma \cup \{\sim\ell\})$;

  if $\mathcal{I}' = \emptyset$ **then** $\mathcal{B} \leftarrow \mathcal{B} \cup \{\ell\}$**else** $\mathcal{I} \leftarrow \mathcal{I} \cap \mathcal{I}'$;

return $\mathtt{bcp}(\Sigma \cup \mathcal{B})$

Backbone Identification (BI): Example

$\Sigma =$

$a \vee b$

$\neg a \vee b$

$\neg b \vee c$

$c \vee d$

$\neg c \vee e \vee f$

$f \vee \neg g$

The backbone of Σ is equal to $B = \{b, c\}$

$\texttt{BI}(\Sigma) =$

b

c

$e \vee f$

$f \vee \neg g$

Properties of BI

- Preserves logical equivalence
- Is confluent and projective
- $\#var(\Sigma) \geq \#var(\texttt{BI}(\Sigma))$
- $\#lit(\Sigma) \geq \#lit(\texttt{BI}(\Sigma))$
- $tw(\Sigma) \geq tw(\texttt{BI}(\Sigma))$

BI: Reduction of the Number of Variables

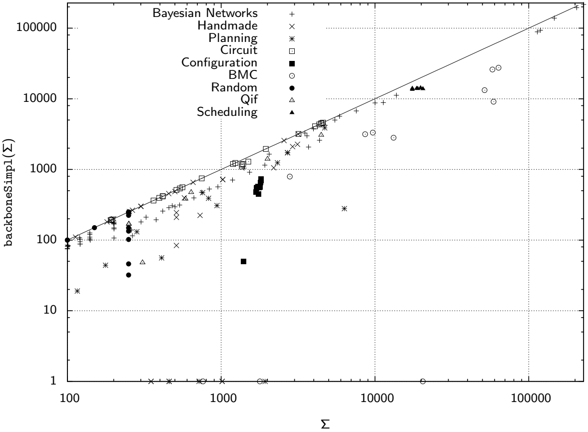


FIGURE: Comparing $\#var(\Sigma)$ with $\#var(\mathtt{BI}(\Sigma))$.

BI: Reduction of the Size

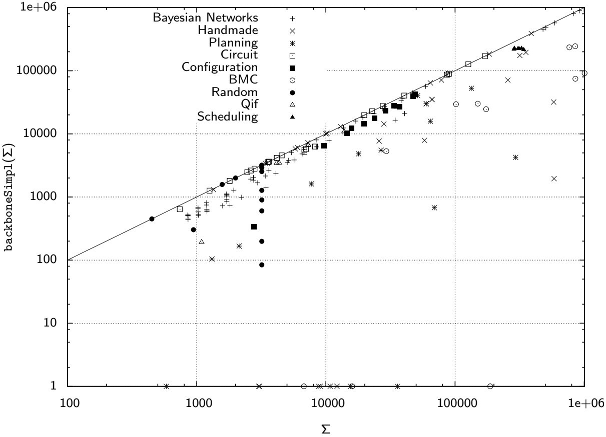


FIGURE: Comparing $\#lit(\Sigma)$ with $\#lit(\mathtt{BI}(\Sigma))$.

BI: Reduction of the Treewidth

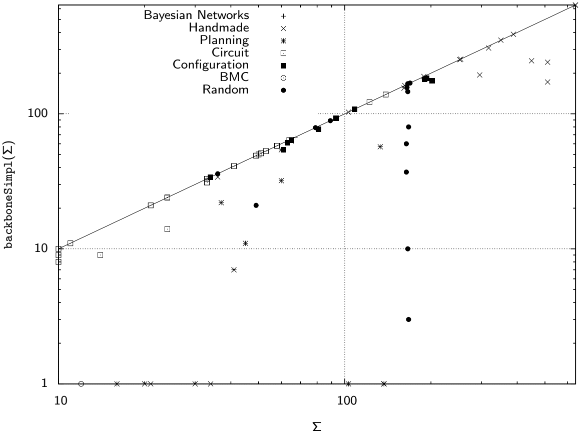


FIGURE: Comparing $tw(\Sigma)$ with $tw(\mathtt{BI}(\Sigma))$.

Overview

Putting the Elementary Preprocessings Together

The `pmc` preprocessor

- Iteration makes sense
- The combination to be chosen depends on what is expected to be preserved
  - *eq* corresponds to the parameter assignment of `pmc` where $optV = optB = optO = 1$ and $optG = 0$. It is equivalence-preserving.
  - $\#eq$ corresponds to the parameter assignment of `pmc` where $optV = optB = optO = 1$ and $optG = 1$. This combination is guaranteed only to preserve the number of models of the input.

The pmc Preprocessor

Algorithm 7: `pmc`

input : a `CNF` formula Σ

output: a `CNF` formula Φ such that $\|\Phi\| = \|\Sigma\|$

$\Phi \leftarrow \Sigma$;
if *optB* **then** $\Phi \leftarrow \mathtt{BI}(\Phi)$;
$i \leftarrow 0$;
while $i < numTries$ **do**
    $i \leftarrow i + 1$;
    if *optO* **then** $\Phi \leftarrow \mathtt{OR}(\Phi)$;
    if *optG* **then** $\Phi \leftarrow \mathtt{XG}(\mathtt{AG}(\mathtt{LE}(\Phi)))$;
    if *optV* **then** $\Phi \leftarrow \mathtt{VI}(\Phi)$;
    if *fixpoint* **then** *break*;
return Φ

eq: Reduction of the Number of Variables

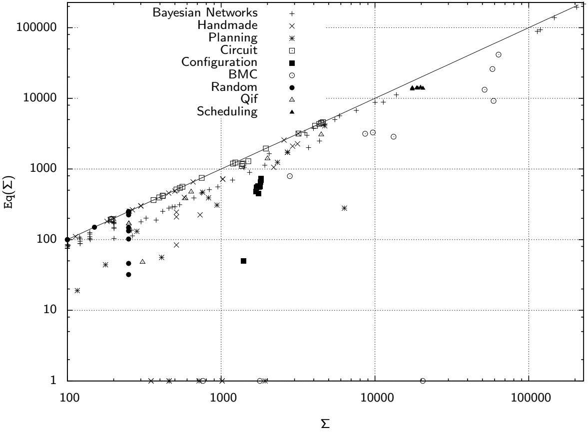


FIGURE: Comparing $\#var(\Sigma)$ with $\#var(eq(\Sigma))$.

eq: Reduction of the Size

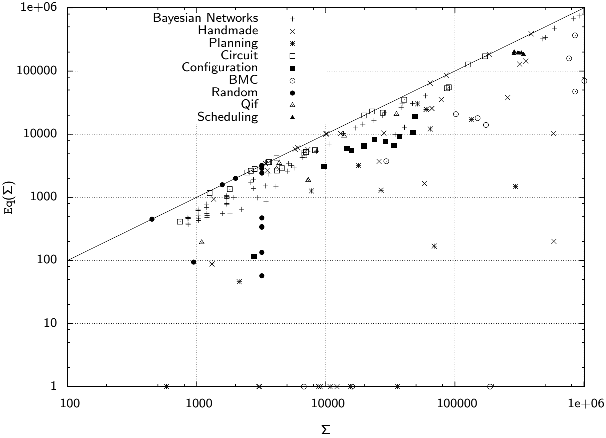


FIGURE: Comparing $\#lit(\Sigma)$ with $\#lit(eq(\Sigma))$.

eq: Reduction of the Treewidth

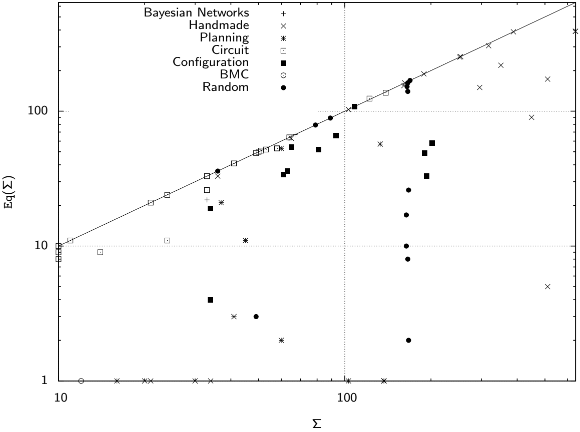


FIGURE: Comparing $tw(\Sigma)$ with $tw(eq(\Sigma))$.

#*eq*: Reduction of the Number of Variables

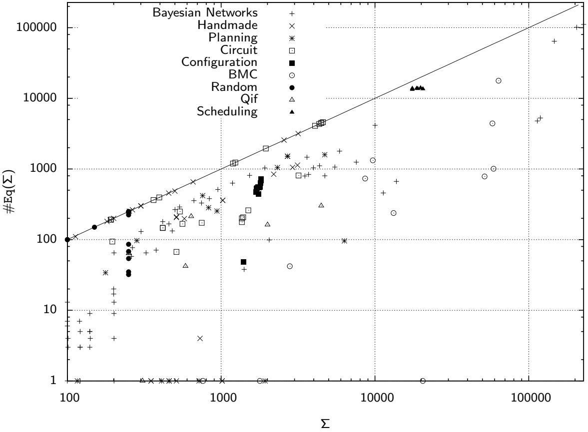


FIGURE: Comparing $\#var(\Sigma)$ with $\#var(\#eq(\Sigma))$.

#*eq*: Reduction of the Size

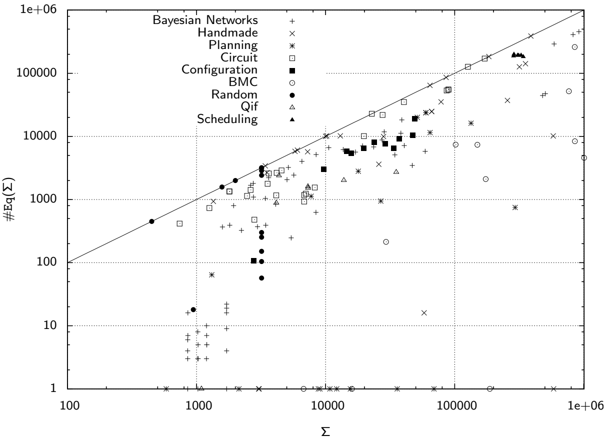


FIGURE: Comparing $\#lit(\Sigma)$ with $\#lit(\#eq(\Sigma))$.

#*eq*: Reduction of the Treewidth

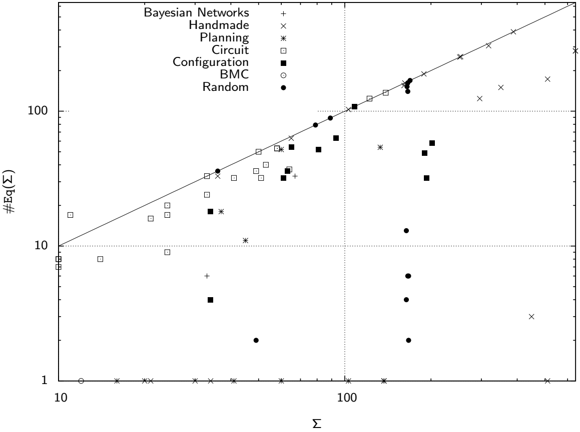


FIGURE: Comparing $tw(\Sigma)$ with $tw(\#eq(\Sigma))$.

Improving Model Counting?

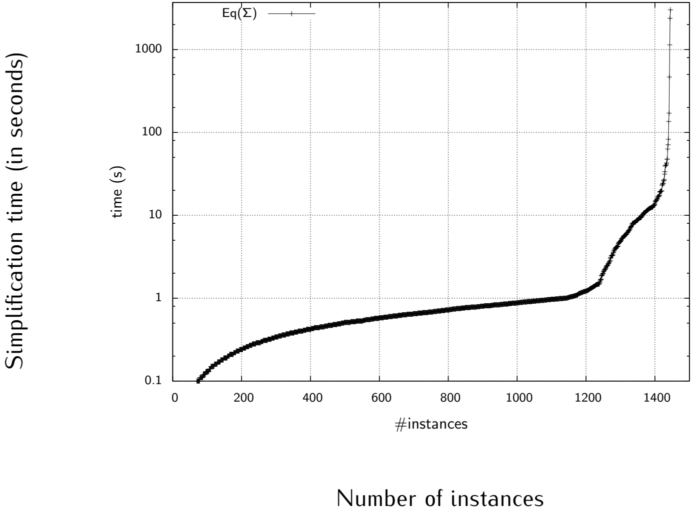
- Reducing $\#var(\Sigma)$, $\#lit(\Sigma)$, and $tw(\Sigma)$ is *a priori* valuable for improving the model counting task (computationally speaking)
- But it could be the case that the preprocessings used are too much time-demanding and that they do not really lead to easier instances but concentrate their difficulty instead...
- Some large-scale experiments are needed to determine whether some improvements are actually achieved
- `Cachet`, `SharpSAT`, `C2D`, and `Dsharp` are used downstream
- `C2D` and `Dsharp` are used as compilers (since the objective is to preserve the information given in the input, only *eq* is considered as an admissible preprocessing combination)

Empirical Setting

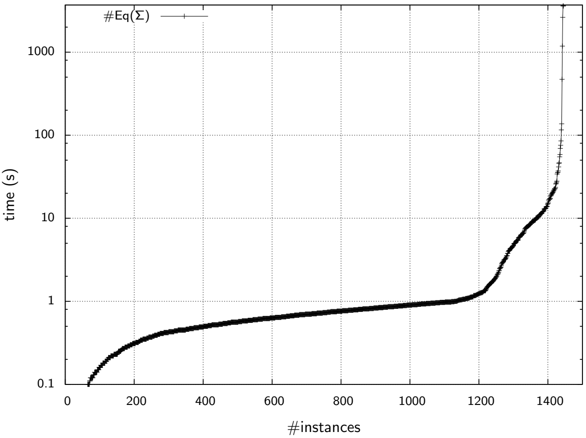
- 1449 `CNF` instances from the SAT LIBrary
- 9 data sets: BMC (18), Circuit (68), Qif (7), Planning (34), Random (105), Scheduling (6), Handmade (58), Configuration (35), Bayesian networks (1118)

- Cluster of Intel Xeon E5-2643 (3.30 GHz) processors with 32 GiB RAM on Linux CentOS
- Time-out =1h
- Memory-out = 7.6 GiB

Efficiency of *eq*



Efficiency of #*eq*



Simplification time (in seconds)

Number of instances

Impact of *eq* on Cachet

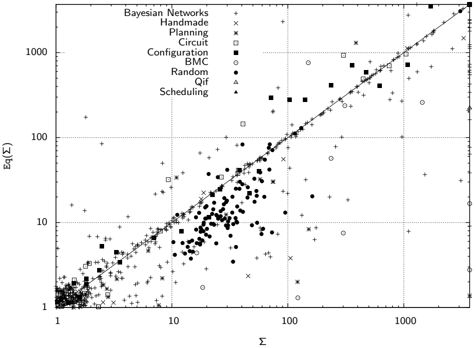


FIGURE: Comparison of the computation times needed to count the number of models of an instance using `Cachet`, when no preprocessing is used vs. when the *eq* combination has been applied first.

Impact of #*eq* on Cachet

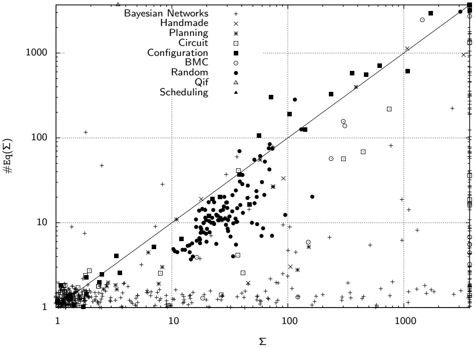


FIGURE: Comparison of the computation times needed to count the number of models of an instance using Cachet, when no preprocessing is used vs. when the #*eq* combination has been applied first.

Impact of *eq* on SharpSAT

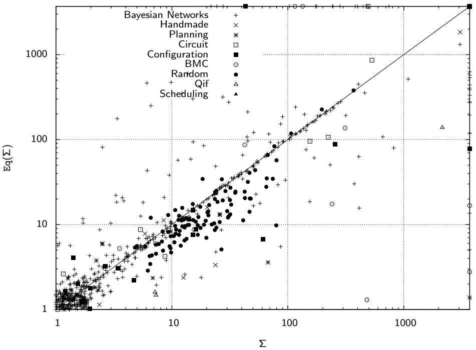


FIGURE: Comparison of the computation times needed to count the number of models of an instance using SharpSAT, when no preprocessing is used vs. when the *eq* combination has been applied first.

Impact of #*eq* on `SharpSAT`

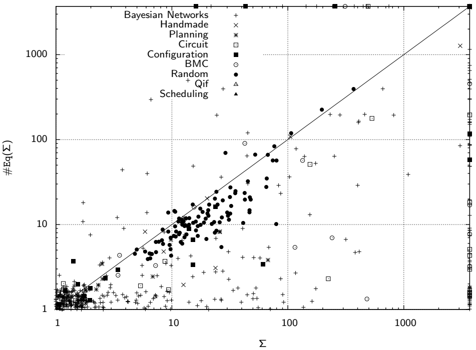


FIGURE: Comparison of the computation times needed to count the number of models of an instance using `SharpSAT`, when no preprocessing is used vs. when the #*eq* combination has been applied first.

Impact of *eq* on C2D (compilation times)

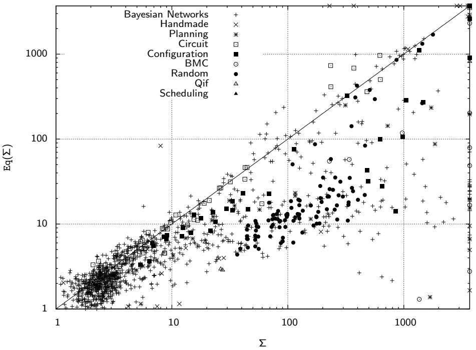


FIGURE: Comparisons of the compilation times of C2D, when no preprocessing is used vs. when the *eq* combination has been applied first.

Impact of *eq* on C2D (sizes of the compiled forms)

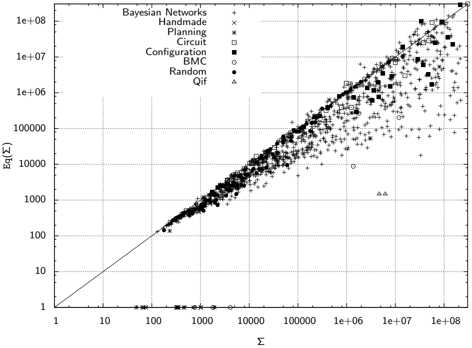


FIGURE: Comparisons of the sizes of the compiled forms obtained using C2D, when no preprocessing is used vs. when the *eq* combination has been applied first.

Impact of *eq* on Dsharp (compilation times)

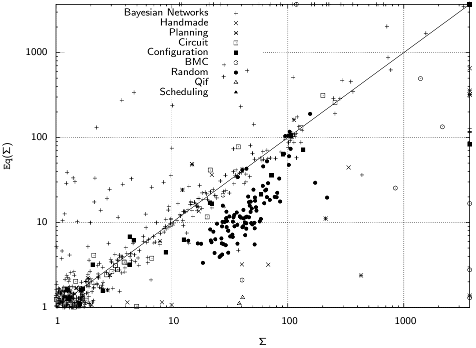


FIGURE: Comparisons of the compilation times of Dsharp, when no preprocessing is used vs. when the *eq* combination has been applied first.

Impact of *eq* on Dsharp (sizes of the compiled forms)

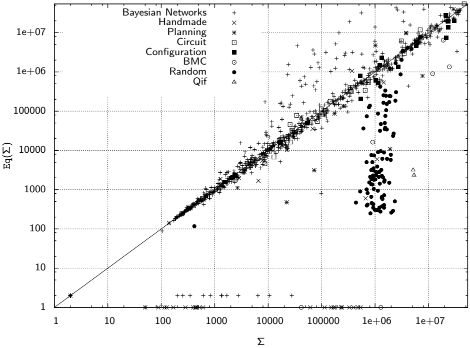


FIGURE: Comparisons of the sizes of the compiled forms obtained using `Dsharp`, when no preprocessing is used vs. when the *eq* combination has been applied first.

Empirical Results

- Empirically, each of *eq* and #*eq* proves to be a useful preprocessing combination (whatever the downstream model counter)
- The gate-detection-and-replacement preprocessings appear as particularly interesting for improving search-based model counters
- However this family of preprocessings is restricted to a small subset of target gates
- Is it possible to do better, and to enlarge this family?

Overview

Limitations of the Gate Detection and Replacement Preprocessings

- The replacement phase **requires gates to be detected**
  - The search space for gates is **huge**
  - The size of a gate can be **huge** as well

Limitations of the Gate Detection and Replacement Preprocessings

- The replacement phase **requires gates to be detected**
  - The search space for gates is **huge**
  - The size of a gate can be **huge** as well

- Identifying "complex gates" is incompatible with the efficiency expected for a preprocessing:
  only "simple" gates are targeted

literal equivalences	$y \leftrightarrow x_1$
AND/OR gates	$y \leftrightarrow (x_1 \wedge \overline{x_2} \wedge x_3)$
XOR gates	$y \leftrightarrow (x_1 \oplus \overline{x_2})$

Overcoming the Limitations (1)

- The (explicit) detection phase can be replaced by an **implicit detection phase**
- There is **no need to identify** f to determine that a gate of the form $y \leftrightarrow f(x_1, \ldots, x_n)$ exists in Σ

Overcoming the Limitations (1)

- The (explicit) detection phase can be replaced by an **implicit detection phase**
- There is **no need to identify** f to determine that a gate of the form $y \leftrightarrow f(x_1, \ldots, x_n)$ exists in Σ

- Let us ask Evert and Alessandro for some help ...

Evert Willem Beth (1908–1964)

Evert Willem Beth (1908–1964)

- Σ **explicitly defines** y in terms of $X = \{x_1, \ldots, x_n\}$ iff there exists a formula $f(x_1, \ldots, x_n)$ over X such that

$$\Sigma \models y \leftrightarrow f(x_1, \ldots, x_n)$$

Evert Willem Beth (1908–1964)

- Σ **explicitly defines** y in terms of $X = \{x_1, \ldots, x_n\}$ iff there exists a formula $f(x_1, \ldots, x_n)$ over X such that

$$\Sigma \models y \leftrightarrow f(x_1, \ldots, x_n)$$

- Σ **implicitly defines** y in terms of $X = \{x_1, \ldots, x_n\}$ iff for every canonical term γ_X over X, we have $\Sigma \wedge \gamma_X \models y$ or $\Sigma \wedge \gamma_X \models \overline{y}$

Evert Willem Beth (1908–1964)

- Σ **explicitly defines** y in terms of $X = \{x_1, \ldots, x_n\}$ iff there exists a formula $f(x_1, \ldots, x_n)$ over X such that

$$\Sigma \models y \leftrightarrow f(x_1, \ldots, x_n)$$

- Σ **implicitly defines** y in terms of $X = \{x_1, \ldots, x_n\}$ iff for every canonical term γ_X over X, we have $\Sigma \wedge \gamma_X \models y$ or $\Sigma \wedge \gamma_X \models \overline{y}$
- Beth's theorem: Σ **explicitly defines** y in terms of X iff Σ **implicitly defines** y in terms of X

Alessandro Padoa (1868–1937)

Padoa's theorem:

Let Σ'_X be equal to Σ where each variable but those of X have been renamed in a uniform way

If $y \notin X$, then Σ (implicitly) defines y in terms of X iff $\Sigma \wedge \Sigma'_X \wedge y \wedge \overline{y'}$ is inconsistent

Alessandro Padoa (1868–1937)

Padoa's theorem:

Let Σ'_X be equal to Σ where each variable but those of X have been renamed in a uniform way
If $y \notin X$, then Σ (implicitly) defines y in terms of X iff $\Sigma \wedge \Sigma'_X \wedge y \wedge \overline{y'}$ is inconsistent

Deciding whether Σ (implicitly) defines y in terms of X is "only" coNP-complete

Overcoming the Limitations (2)

- There is **no need to identify** f to determine that a gate of the form $y \leftrightarrow f(x_1, \ldots, x_n)$ exists in Σ

Overcoming the Limitations (2)

- There is **no need to identify** f to determine that a gate of the form $y \leftrightarrow f(x_1, \ldots, x_n)$ exists in Σ
  - **Gate detection = Explicit definability**

Overcoming the Limitations (2)

- There is **no need to identify** f to determine that a gate of the form $y \leftrightarrow f(x_1, \ldots, x_n)$ exists in Σ
  - **Gate detection = Explicit definability**
  - **Explicit definability = Implicit definability (Beth's theorem)**

Overcoming the Limitations (2)

- There is **no need to identify** f to determine that a gate of the form $y \leftrightarrow f(x_1, \ldots, x_n)$ exists in Σ
  - **Gate detection = Explicit definability**
  - **Explicit definability = Implicit definability (Beth's theorem)**
  - **One call to a SAT solver** is enough to decide whether Σ defines y in terms of $\{x_1, \ldots, x_n\}$ (thanks to Padoa's theorem)

Overcoming the Limitations (2)

- There is **no need to identify** f to determine that a gate of the form $y \leftrightarrow f(x_1, \ldots, x_n)$ exists in Σ
  - **Gate detection = Explicit definability**
  - **Explicit definability = Implicit definability (Beth's theorem)**
  - **One call to a SAT solver** is enough to decide whether Σ defines y in terms of $\{x_1, \ldots, x_n\}$ (thanks to Padoa's theorem)
- There is **no need to identify** f to compute $\Sigma[y \leftarrow f(x_1, \ldots, x_n)]$

Overcoming the Limitations (2)

- There is **no need to identify** f to determine that a gate of the form $y \leftrightarrow f(x_1, \ldots, x_n)$ exists in Σ
  - **Gate detection = Explicit definability**
  - **Explicit definability = Implicit definability (Beth's theorem)**
  - **One call to a SAT solver** is enough to decide whether Σ defines y in terms of $\{x_1, \ldots, x_n\}$ (thanks to Padoa's theorem)

- There is **no need to identify** f to compute $\Sigma[y \leftarrow f(x_1, \ldots, x_n)]$
  - The replacement phase can be replaced by an **output variable elimination phase:** if $y \leftrightarrow f(x_1, \ldots, x_n)$ is a gate of Σ, then

$$\Sigma[y \leftarrow f(x_1, \ldots, x_n)] \equiv \exists y.\Sigma$$

The $\mathtt{B} + \mathtt{E}$ Preprocessor

A two-step preprocessing

- "Detection = Bipartition":
  compute a **definability bipartition** $\langle I, O \rangle$ of Σ such that $I \cup O = Var(\Sigma)$, $I \cap O = \emptyset$, and Σ defines every variable $o \in O$ in terms of I

The $\mathtt{B}+\mathtt{E}$ Preprocessor

A two-step preprocessing

- "Detection = Bipartition":
  compute a **definability bipartition** $\langle I, O\rangle$ of Σ such that $I \cup O = Var(\Sigma)$, $I \cap O = \emptyset$, and Σ defines every variable $o \in O$ in terms of I
- "Replacement = Elimination":
  compute $\exists E.\Sigma$ for $E \subseteq O$

The B + E Preprocessor

A two-step preprocessing

- "Detection = <u>B</u>ipartition":
  compute a **definability bipartition** $\langle I, O \rangle$ of Σ such that $I \cup O = Var(\Sigma)$, $I \cap O = \emptyset$, and Σ defines every variable $o \in O$ in terms of I
- "Replacement = <u>E</u>limination":
  compute $\exists E.\Sigma$ for $E \subseteq O$

- Steps B and E of B + E can be tuned in order to keep the preprocessing phase **light from a computational standpoint**

Back to GDR

$$\Sigma = \quad \begin{array}{l} \overline{x} \vee u \vee v \\ \overline{x} \vee \overline{y} \vee u \\ \overline{x} \vee \overline{z} \vee u \\ x \vee \overline{u} \\ y \vee z \vee \overline{u} \end{array} \qquad\qquad u \leftrightarrow (x \wedge (y \vee z))$$

Σ defines u in terms of $\{x, y, z\}$

Detecting u as an Output Variable and Eliminating it

Identification:
$\Sigma \wedge \Sigma'_{\{x,y,z\}} \wedge u \wedge \overline{u'}$ is inconsistent

$\overline{x} \vee u \vee v$
$\overline{x} \vee \overline{y} \vee u$
$\overline{x} \vee \overline{z} \vee u$
$x \vee \overline{u}$
$y \vee z \vee \overline{u}$
$\overline{x} \vee u' \vee v'$
$\overline{x} \vee \overline{y} \vee u'$
$\overline{x} \vee \overline{z} \vee u'$
$x \vee \overline{u'}$
$y \vee z \vee \overline{u'}$
u
$\overline{u'}$

Detecting u as an Output Variable and Eliminating it

Identification:
$\Sigma \wedge \Sigma'_{\{x,y,z\}} \wedge u \wedge \overline{u'}$ is inconsistent

$\overline{x} \vee u \vee v$
$\overline{x} \vee \overline{y} \vee u$
$\overline{x} \vee \overline{z} \vee u$
$x \vee \overline{u}$
$y \vee z \vee \overline{u}$
$\overline{x} \vee u' \vee v'$
$\overline{x} \vee \overline{y} \vee u'$
$\overline{x} \vee \overline{z} \vee u'$
$x \vee \overline{u'}$
$y \vee z \vee \overline{u'}$
u
$\overline{u'}$

Elimination:
computing resolvents over u

$\overline{x} \vee v \vee x$	valid
$\overline{x} \vee v \vee y \vee z$	
$\overline{x} \vee \overline{y} \vee x$	valid
$\overline{x} \vee \overline{y} \vee y \vee z$	valid
$\overline{x} \vee \overline{z} \vee x$	valid
$\overline{x} \vee \overline{z} \vee y \vee z$	valid

Detecting u as an Output Variable and Eliminating it

Identification:
$\Sigma \wedge \Sigma'_{\{x,y,z\}} \wedge u \wedge \overline{u'}$ is inconsistent

$\overline{x} \vee u \vee v$
$\overline{x} \vee \overline{y} \vee u$
$\overline{x} \vee \overline{z} \vee u$
$x \vee \overline{u}$
$y \vee z \vee \overline{u}$
$\overline{x} \vee u' \vee v'$
$\overline{x} \vee \overline{y} \vee u'$
$\overline{x} \vee \overline{z} \vee u'$
$x \vee \overline{u'}$
$y \vee z \vee \overline{u'}$
u
$\overline{u'}$

Elimination:
computing resolvents over u

$\overline{x} \vee v \vee x$	valid
$\overline{x} \vee v \vee y \vee z$	
$\overline{x} \vee \overline{y} \vee x$	valid
$\overline{x} \vee \overline{y} \vee y \vee z$	valid
$\overline{x} \vee \overline{z} \vee x$	valid
$\overline{x} \vee \overline{z} \vee y \vee z$	valid

$\|\Sigma\| = \|\overline{x} \vee v \vee y \vee z\| = 15$

Tuning the Computational Effort

Both steps $\mathtt{B}$ and $\mathtt{E}$ of $\mathtt{B} + \mathtt{E}$ can be tuned in order to keep the preprocessing phase **light from a computational standpoint**

- It is not necessary to determine a definability bipartition $\langle I, O \rangle$ with $|I|$ minimal
  $\Rightarrow$ $\mathtt{B}$ is a **greedy algorithm** (one definability test per variable)
  $\Rightarrow$ Only the minimality of I for $\subseteq$ is guaranteed

Tuning the Computational Effort

Both steps $\mathtt{B}$ and $\mathtt{E}$ of $\mathtt{B} + \mathtt{E}$ can be tuned in order to keep the preprocessing phase **light from a computational standpoint**

- It is not necessary to determine a definability bipartition $\langle I, O \rangle$ with $|I|$ minimal
  $\Rightarrow$ $\mathtt{B}$ is a **greedy algorithm** (one definability test per variable)
  $\Rightarrow$ Only the minimality of I for $\subseteq$ is guaranteed

- It is not necessary to eliminate in Σ every variable of O but focusing on a subset $E \subseteq O$ is enough
  $\Rightarrow$ Eliminating every output variable could lead to an **exponential blow up**
  $\Rightarrow$ The elimination of $y \in O$ is committed only if $|\Sigma|$ after the elimination step and some additional preprocessing (occurrence simplification and vivification) remains **small enough**

Experiments

Objectives:

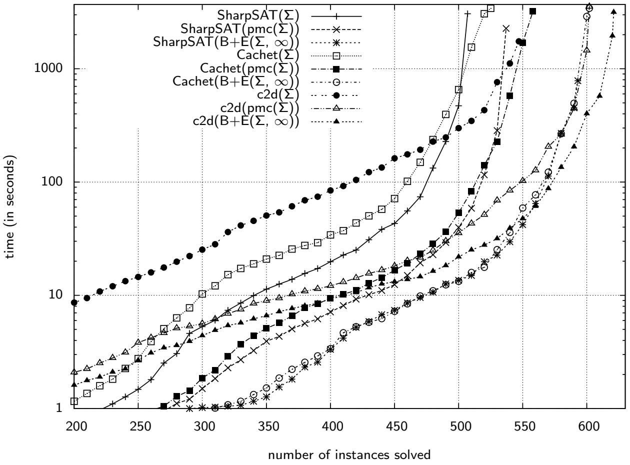
- Evaluating the computational benefits offered by $\texttt{B} + \texttt{E}$ when used upstream to state-of-the-art model counters:
  - the search-based model counter `Cachet`
  - the search-based model counter `SharpSAT`
  - the compilation-based model counter `C2D`

Experiments

Objectives:

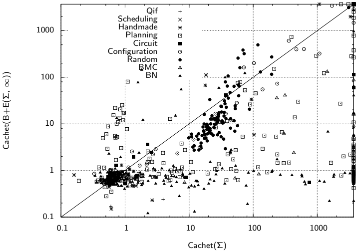
- Evaluating the computational benefits offered by $\mathtt{B} + \mathtt{E}$ when used upstream to state-of-the-art model counters:
  - the search-based model counter `Cachet`
  - the search-based model counter `SharpSAT`
  - the compilation-based model counter `C2D`
- Comparing the benefits offered by $\mathtt{B} + \mathtt{E}$ with those offered by our previous preprocessor `pmc` (based on gate identification and replacement) or with no preprocessing

Empirical Results

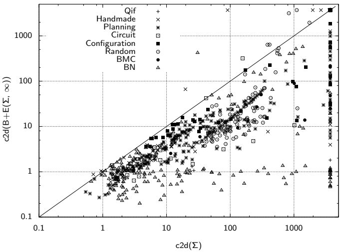


Empirical Results

$\mathtt{B} + \mathtt{E}$ vs. no preprocessing



(a) $\mathtt{B} + \mathtt{E}$+Cachet vs. Cachet

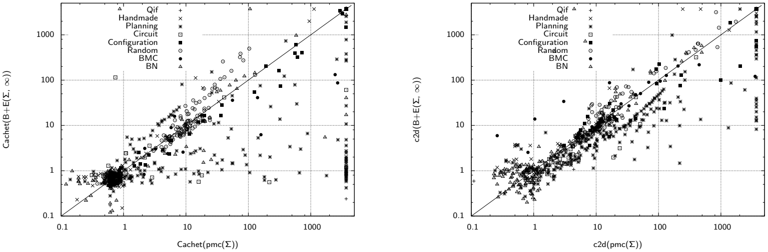


(b) $\mathtt{B} + \mathtt{E}$+C2D vs. C2D

FIGURE: Model counting time reductions achieved by $\mathtt{B} + \mathtt{E}$ vs. no preprocessing

Empirical Results

$\mathtt{B} + \mathtt{E}$ vs. `pmc`



(a) $\mathtt{B} + \mathtt{E}$+`Cachet` vs. `pmc`+`Cachet` (b) $\mathtt{B} + \mathtt{E}$+`C2D` vs. `pmc`+`C2D`

FIGURE: Model counting time reductions achieved by $\mathtt{B} + \mathtt{E}$ vs. `pmc`

Empirical Results

- The experiments clearly show the benefits offered by `B + E`
- `B + E` appears typically as a better preprocessor than `pmc` since it leads typically to improved performances (smaller computation times)

References (for further reading)

G. Audemard, and L. Simon. Predicting learnt clauses quality in modern SAT solver. IJCAI'09, pages 399–404, 2009.

R.A. Aziz, G. Chu, C.J. Muise, and P.J. Stuckey. #∃sat: Projected model counting. SAT'15, pages 121–137, 2015.

F. Bacchus, S. Dalmao, and T. Pitassi. Algorithms and complexity results for #sat and Bayesian inference. FOCS'03, pages 340–351, 2003.

F. Bacchus, and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. SAT'04, pages 341–355, 2004.

E. Beth. On Padoa's method in the theory of definition. Indagationes mathematicae 15:330–339, 1953.

A. Biere. Lingeling essentials, A tutorial on design and implementation aspects of the SAT solver Lingeling. POS'14, page 88, 2014.

Y. Boufkhad, and O. Roussel. Redundancy in Random SAT Formulas. AAAI/IAAI'00, pages 273-278, 2000.

O. Čepek, P. Kučera, and P. Savický. Boolean functions with a simple certificate for CNF complexity. Discrete Applied Mathematics 160(4-5): 365-382, 2012.

A. Darwiche. Decomposable negation normal form. Journal of the ACM, 48(4):608–647, 2001.

A. Darwiche. New advances in compiling cnf into decomposable negation normal form. ECAI'04, pages 328–332, 2004.

M. Gebser, B. Kaufmann, and T. Schaub. Solution enumeration for projected boolean search problems. CPAIOR'09, pages 71–86, 2009.

C.P. Gomes, A. Sabharwal, and B. Selman. Model counting. Handbook of Satisfiability, pages 633–654, 2009.

G. Gottlob, and C.G. Fermüller. Removing redundancy from a clause. Artificial Intelligence 61:263–289, 1993.

H. Hanand, F. Somenzi. Alembic: An efficient algorithm for CNF preprocessing. DAC'07, pages 582–587, 2007.

M. Heule, M. Järvisalo, and A. Biere. Clause elimination procedures for CNF formulas. LPAR'10, pages 357–371, 2010.

M. Heule, M. Järvisalo, and A. Biere.Covered clause elimination. LPAR'10, pages 41–46, 2010.

References (for further reading)

M. Heule, M. Järvisalo, and A. Biere.Efficient cnf simplification based on binary implication graphs. SAT'11, pages 201–215, 2011.

M. Heule, M. Järvisalo, F. Lonsing, M. Seidl, and A. Biere. Clause elimination for SAT and QSAT. J. Artif. Intell. Res. (JAIR), 53:127-168, 2015.

M. Järvisalo, A. Biere, and M. Heule. Simulating circuit-level simplifications on CNF. Journal of Automated Reasoning, 49(4):583–619, 2012.

V. Klebanov, N. Manthey, and C. J. Muise. Sat-based analysis and quantification of information flow in programs. QUEST'13, pages 177–192, 2013.

J.-M. Lagniez, and P. Marquis. On Preprocessing Techniques and Their Impact on Propositional Model Counting. J. Autom. Reasoning 58(4): 413-481, 2017.

J.-M. Lagniez, E. Lonca, and P. Marquis. Improving Model Counting by Leveraging Definability. IJCAI'16, pages 751-757, 2016

J. Lang, and P. Marquis. On propositional definability. Artificial Intelligence 172 (8-9):991–1017, 2008.

P. Liberatore. Redundancy in logic I: CNF propositional formulae. Artif. Intell. 163(2): 203-232, 2005.

I. Lynce, and J. Marques-Silva. Probing-based preprocessing techniques for propositional satisfiability. ICTAI'03, pages 105–110, 2003.

N. Manthey. Coprocessor 2.0 - A flexible CNF simplifier - (tool presentation). SAT'12, pages 436–441, 2012.

N. Manthey. Solver description of RISS2.0 and PRISS2.0. Technical report, TU Dresden, Knowledge Representation and Reasoning, 2012.

Ch.J. Muise, Sh.A. McIlraith, J.Ch. Beck, and E.I. Hsu. Dsharp: Fast d-DNNF compilation with sharpSAT. AI'12, pages 356–361, 2012.

References (for further reading)

R. Ostrowski, É. Grégoire, B. Mazure, and L. Saïs. Recovering and exploiting structural knowledge from CNF formulas. CP'02, pages 185–199, 2002.

A. Padoa. Essai d'une théorie algébrique des nombres entiers, précédé d'une introduction logique à une théorie déductive quelconque. Bibliothèque du Congrès International de Philosophie, Paris, pages 309–365, 1903.

C. Piette, Y. Hamadi, and L. Saïs. Vivifying propositional clausal formulae. ECAI'08, pages 525–529, 2008.

D. A. Plaisted, and S. Greenbaum. A Structure-Preserving Clause Form Translation. J. Symb. Comput. 2(3): 293-304, 1986.

M. Samer, and S. Szeider. Algorithms for propositional model counting, J. Discrete Algorithms 8 (1):50–64, 2010.

T. Sang, F. Bacchus, P. Beame, H.A. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. SAT'04, 2004.

S. Subbarayan, and D. Pradhan. NiVER: Non increasing variable elimination resolution for preprocessing SAT instances. SAT'04, pages 276–291, 2004.

S. Toda. PP is as hard as the polynomial-time hierarchy. SIAM Journal on Computing 20 (5):865–877, 1991 .

M. Thurley. sharpSAT - counting models with advanced component caching and implicit BCP. SAT'06, pages 424–429, 2006.

G. Tseitin. On the complexity of derivation in propositional calculus. Steklov Mathematical Institute, Chapter "Structures in Constructive Mathematics and Mathematical Logic", pp. 115–125, 1968.

Ch. Umans. The Minimum Equivalent DNF Problem and Shortest Implicants. J. Comput. Syst. Sci. 63(4): 597-611, 2001.

H. Zhang and M.E. Stickel. An efficient algorithm for unit propagation. ISAIM'96, pages 166–169, 1996.

NP-Preprocessing

Pierre Marquis

CRIL, U. Artois & CNRS,
Institut Universitaire de France
France