Noname manuscript No.
(will be inserted by the editor)

On Preprocessing Techniques and their Impact on Propositional Model Counting

Jean-Marie Lagniez · Pierre Marquis

Received: date / Accepted: date

Abstract This paper is concerned with preprocessing techniques for propositional model counting. We have considered several elementary preprocessing techniques: backbone identification, occurrence reduction, vivification, as well as equivalence, AND and XOR gate identification and replacement. All those techniques have been implemented in a preprocessor `pmc`, freely available on the Web. In order to assess the benefits which can be gained by taking advantage of `pmc`, we performed many experiments, based on benchmarks coming from several data sets. More precisely, we made a differential evaluation of each elementary preprocessing technique in order to evaluate its impact on the number of variables of the instance, its size, as well as the treewidth of its primal graph. We also considered two combinations of preprocessings: eq, based on equivalence-preserving techniques only, and $\#eq$, which additionally exploits techniques preserving only the number of models. Several approaches to model counting have also been considered downstream in our experiments: "direct" model counters, including the exact ones `Cachet`, `sharpSAT`, and an approximate one `SampleCount`, as well as the compilation-based model counters `C2D`, `Dsharp`, `SDD` and `cnf2obdd` have been used. The experimental results we have obtained show that each elementary preprocessing technique is useful, and that some synergetic effects can be achieved by combining them.

Keywords Propositional model counting · Preprocessing

Jean-Marie Lagniez
CRIL-CNRS and Université d'Artois
Lens, France
E-mail: lagniez@cril.univ-artois.fr

Pierre Marquis
CRIL-CNRS and Université d'Artois
Lens, France
E-mail: marquis@cril.univ-artois.fr

1 Introduction

Preprocessing a propositional formula basically consists in turning it into another propositional formula, while preserving some property, for instance its satisfiability. It proves useful when the problem under consideration (e.g., the satisfiability issue) can be solved more efficiently when the input formula has been first preprocessed (while taking account for the preprocessing time in the global solving time). Some preprocessing techniques are nowadays acknowledged as valuable for SAT solving and QBF solving (see e.g., [5,54,38,21,46,26,27,32,29]), leading to computational improvements. Let us mention among others the following techniques:

- Vivification (VIV) [46], and a light form of it, that we call Occurrence Elimination (OE),
- Gate Detection and Replacement (GDR) [5,44,21],
- Pure Literal Elimination (PLE) [17,12],
- Variable Elimination (VE) [18,54],
- Blocked Clause Elimination (BCE) [35],
- Covered Clause Elimination (CCE) [28,30],
- Failed Literal Elimination (FLE) [23],
- Self-Subsuming Resolution (SSR) [21],
- Hidden Literal Elimination (HLE) [29],
- Subsumption Elimination (SE) [12,37,7,21],
- Hidden Subsumption Elimination (HSE) [27],
- Asymmetric Subsumption Elimination (ASE) [27,30],
- Tautology Elimination (TE) [12,37,7,21],
- Hidden Tautology Elimination (HTE) [27]
- Asymmetric Tautology Elimination (ATE) [27,30].

As such, many of those techniques are now embodied in some state-of-the-art SAT solvers, like `Glucose` [4] which takes advantage of the `SatELite` preprocessor [21], `Lingeling` [6] which has an internal preprocessor, and `Riss` [40] which takes advantage of the `Coprocessor` preprocessor [39].

In this paper, we focus on preprocessing techniques for *propositional model counting*, i.e., the problem which consists in determining the number of truth assignments satisfying a given propositional formula (typically represented into conjunctive normal form – `CNF`). Model counting and its direct generalization, weighted model counting,[1] are central to many AI problems including probabilistic inference (see e.g., [51,13,2]) and forms of planning (see e.g., [45,20]). However, model counting is a computationally demanding task (it is #P-complete [57] even for monotone 2-`CNF` formulae and Horn 2-`CNF` formulae), and hard to approximate (it is NP-hard to approximate the number of models of a formula with n variables within $2^{n^{1-\epsilon}}$ for $\epsilon > 0$ [48]). Especially, it is harder (both in theory and in practice) than SAT.

[1] In weighted model counting (WMC), each literal is associated with a real number, the weight of an interpretation is the product of the weights of the literals it sets to true, and the weight of a formula is the sum of the weights of its models. Accordingly, WMC amounts to model counting when each literal has weight 1.

Focusing on model counting instead of satisfiability has some important impacts on the preprocessings which ought to be considered. On the one hand, preserving satisfiability is not enough for ensuring that the number of models does not change. Thus, some efficient preprocessing techniques considered for SAT must be let aside; this includes the *pure literal elimination rule (PLE)* (removing every clause from the input CNF formula which contains a pure literal, i.e., a literal appearing with the same polarity in the whole formula), and more importantly the *variable elimination rule (VE)* (replacing in the input CNF formula all the clauses containing a given variable x by the set of all their resolvents over x), the *blocked clause elimination rule (BCE)* [35] (removing every clause containing a literal such that every resolvent obtained by resolving on it is a valid clause) and more generally the *covered clause elimination rule (CCE)* (roughly, removing clauses which are rendered blocked via the addition of some literals in an equivalence-preserving way) [30]. Indeed, while each of these preprocessings preserves the satisfiability of the input formula, none of them is guaranteed to preserve its number of models. On the other hand, the high complexity of model counting allows for considering more aggressive, yet time-consuming, preprocessing techniques than the ones considered when dealing with the satisfiability issue. For instance, it can prove useful to compute the backbone of the given instance Σ before counting its models (the backbone of Σ is the set of all literals implied by Σ); however, while deciding whether $\Sigma \models \ell$ for every literal ℓ over the variables of Σ is enough to determine the satisfiability of Σ, it is also more computationally demanding in practice. Thus it would not make sense to consider backbone detection as a preprocessing for SAT. Finally, since many elementary preprocessings exist, it makes sense to study how to combining them. Preprocessings are more or less efficient, where the efficiency of a technique can be defined (for instance) as the relative ability to reduce the input clauses (hence favoring propagation) or to remove them (hence reducing the size of the instance). Of course, it makes sense to select only the most efficient preprocessing techniques. Furthermore, the ordering according to which elementary preprocessings are combined appears as very relevant.

Another important aspect for the choice of candidate preprocessing techniques is the nature of the model counter to be used downstream. If a "direct" model counter is exploited, then preserving the number of models is enough. Contrastingly, if a compilation-based approach is used (i.e., the input formula is first turned into an equivalent compiled form during an off-line phase, and this compiled form supports efficient conditioning and model counting), preserving equivalence (which is more demanding than preserving the number of models) is mandatory. Furthermore, when a compilation-based approach is considered, the compilation time, i.e., the time spent to compute a compiled form, is not as significant as the size of the compiled form, provided that it can be balanced by sufficiently many on-line queries. Hence it is relevant to focus on the impact of the preprocessing on the size of the compiled form (and not only on the time needed to compute it) when considering a compilation-based model counter.

In this paper, we have studied the adequacy and the performance of several elementary preprocessings for model counting: backbone identification (BI) occurrence elimination (OE), vivification (VIV), as well as equivalence, AND and XOR gate identification and replacement (GDR). The three former techniques preserve equiv-

alence, and as such they can be used whatever the downstream approach to model counting (or to weighted model counting); contrastingly, the three latter ones preserve the number of models of the input, but not equivalence in the general case. We have implemented a preprocessor `pmc` for model counting which implements all those techniques. Starting with a `CNF` formula Σ, it returns a `CNF` formula $\texttt{pmc}(\Sigma)$ which is equivalent or has the same number of models as Σ (depending on the chosen elementary preprocessings which are used).

In order to evaluate the gain which could be offered by exploiting those preprocessing techniques for model counting, we performed quite intensive experiments on a huge number of benchmarks, coming from many families. Our evaluation has been threefold:

- We first evaluated each elementary preprocessing p by focusing on three (ordered) pairs of scores, intended to measure the "simplification effect" which results from applying p. The first coordinate of each pair of scores concerns the input `CNF` formula (before the application of p), while the second coordinate concerns the output `CNF` formula (once p has been applied). Each score can be considered as a measure of the "complexity" of the instance. For each `CNF` instance Σ, we considered the number of variables of Σ (measured as $\#var(\Sigma)$, the cardinality of $Var(\Sigma)$), the size of Σ measured as $\#lit(\Sigma)$, the number of literals occurring in Σ, together with the value of the treewidth $tw(\Sigma)$ of the primal graph associated with Σ. For each p, we first considered both the best and the worst case scenarios; so as to figure out what happens in practice, we also computed the three pairs of scores for 182 benchmarks (encoded as `CNF` formulae) coming from 9 data sets. Then, in order to determine whether synergetic effects may emerge from combining several elementary preprocessings, we also considered two combinations of them: the first one, eq, gathers equivalence-preserving preprocessing only, and the second one, $\#eq$, also takes advantage of preprocessings which are only guaranteed to preserve the number of models. We evaluated the impact of those two combinations by considering again how the three scores above evolve when each combination of preprocessings is applied.
- In a second step, we evaluated the impact of each elementary preprocessing p by coupling it with a model counter. Several approaches to model counting have been considered downstream; they consist of "direct" model counters:
  - the exact model counter `Cachet` (`www.cs.rochester.edu/~kautz/Cachet/index.htm`) [50],
  - the exact model counter `sharpSAT` (`sites.google.com/site/marcthurley/sharpsat`) [55],
  - the approximate model counter `SampleCount` (`www.cs.cornell.edu/~sabhar/#software`) [25],

  and of compilation-based model counters:
  - the top-down `C2D` compiler targeting the Decision-`DNNF` language[2] (`reasoning.cs.ucla.edu/c2d/`) [14, 15],

[2] `d-DNNF` is the language of deterministic, decomposable negation normal form formulae; this language supports the model counting query in polynomial time [14].

  - the top-down Dsharp compiler targeting as well the Decision-DNNF language (www.haz.ca/research/dsharp/) [43],
  - the bottom-up SDD compiler targeting the SDD language – a subset of d-DNNF – (reasoning.cs.ucla.edu/sdd/) [16],
  - the top-down cnf2obdd compiler targeting the OBDD language – a subset of Decision-DNNF – (www.sd.is.uec.ac.jp/toda/code/cnf2obdd.html) [56], following the approach presented in [31].

  As explained before, only equivalence-preserving techniques (including the eq combination) have been exploited when the compilation-based approaches have been considered. For each pair formed by a preprocessing technique p and a model counter, we compared the numbers of benchmarks (over 182) solved within a time limit of 1h per instance, when the preprocessing was used, and when it was not.
- Finally, we evaluated the benefits offered by the two combinations eq and $\#eq$ on a much larger scale, by considering 1449 CNF instances from 9 data sets. For the direct model counters, we compared for each instance the time needed to solve it (i.e., to compute the number of models) when no preprocessing is used, with the time needed to solve it when eq (resp. $\#eq$) has been applied first (of course, the preprocessing time is part of the global solving time), and we also compared eq with $\#eq$. For the compilation-based model counters, we performed a similar comparison, yet focusing on the compilation times and the sizes of the resulting compiled forms, computed as the number of arcs in the obtained DAG. We also computed the total number of instances "solved" by each approach within a time limit of 1h per instance.

The run-time code of our preprocessor pmc and some benchmarks considered in our experiments, are available from www.cril.fr/KC/.

The rest of the paper is organized as follows. Section 2 gives some formal preliminaries. Section 3 presents all the elementary preprocessing techniques we have considered, as well as the two combinations eq and $\#eq$. For each technique p, the way the values of the $\#var$ measure, the $\#lit$ measure, and the tw measure evolve when p is applied is investigated, both from the theory side and from the practical side. Section 4 discusses the benefits which can be gained by taking advantage of p for the family of model counters presented above. Section 5 focuses on the two combinations eq and $\#eq$ and reports experimental results showing their impact on the time needed to count models (for the "direct" model counters), and on both the compilation times and the sizes of the compiled forms (for the compilation-based model counters). We also investigate the possible correlations between the reductions of the input instances in term of $\#var$, $\#lit$ and tw achieved by any of those two combinations, and the subsequent benefits obtained in terms of the times needed for model counting (and both the compilation times and the sizes of the compiled representations for compilation-based model counters). Section 6 discusses other related work; especially, we show that the clause reduction techniques listed at the beginning of this introductive section, but not implemented in our preprocessor pmc, are less efficient than occurrence elimination w.r.t. clause reduction, or less efficient than vivification

w.r.t. clause elimination. This explains why all those techniques have not been incorporated into pmc. Finally, Section 7 concludes the paper.

2 Formal Preliminaries

We consider a propositional language $PROP_{PS}$ defined in the usual way from a finite set PS of propositional symbols and a set of connectives including negation, conjunction, disjunction, equivalence and XOR. Formulae from $PROP_{PS}$ are denoted using Greek letters and Latin letters are used for denoting variables and literals. For every literal ℓ, $var(\ell)$ denotes the variable x of ℓ (i.e., $var(x) = x$ and $var(\neg x) = x$), and $\sim\ell$ denotes the complementary literal of ℓ (i.e., for every variable x, $\sim x = \neg x$ and $\sim\neg x = x$).

$Var(\Sigma)$ is the set of propositional variables occurring in Σ. $|\Sigma|$ denotes the size of Σ. A CNF formula Σ is a conjunction of clauses, where a clause is a disjunction of literals. Every CNF is viewed as a set of clauses, and every clause is viewed as a set of literals. For any clause α, $\sim\alpha$ denotes the term (also viewed as a set of literals) whose literals are the complementary literals of the literals of α. $Lit(\Sigma)$ denotes the set of all literals occurring in a CNF formula Σ.

The *primal graph* of a CNF formula Σ is the (undirected) graph (V, E) where $V = Var(\Sigma)$ and $\{v_i, v_j\} \in E$ iff there exists a clause α of Σ such that $\{v_i, v_j\} \subseteq Var(\alpha)$.

The *treewidth* of a graph (V, E) [47] is the size (minus 1) of the largest vertex set in a tree decomposition of (V, E). It is also equal to the size (minus 1) of the largest clique in a chordal completion of the graph.

$PROP_{PS}$ is interpreted in a classical way. Every interpretation $\mathcal{I}$ (i.e., a mapping from PS to $\{0, 1\}$) is also viewed as a (conjunctively interpreted) set of literals. $\|\Sigma\|$ denotes the number of models of Σ over $Var(\Sigma)$. The model counting problem consists in computing $\|\Sigma\|$ given Σ.

Generally speaking, a *propositional preprocessing* is an algorithm p mapping any formula Σ from $PROP_{PS}$ to a formula $p(\Sigma)$ from $PROP_{PS}$. In the following we focus on preprocessings mapping CNF formulae to CNF formulae.

We also make use of the following notations in the rest of the paper:

- solve(Σ) returns $\emptyset$ if the CNF formula Σ is unsatisfiable, and solve(Σ) returns a model of Σ otherwise.
- bcp denotes a Boolean Constraint Propagator [58], which is a key component of many preprocessors. bcp(Σ) returns $\{\emptyset\}$ if there exists a unit refutation from the clauses of the CNF formula Σ, and it returns the set of literals (unit clauses) which are derived from Σ using unit propagation in the remaining case. Its worst-case time complexity is linear in the input size but quadratic when the set of clauses under consideration is implemented using watched literals [58,42]. As a side effect, bcp "virtually" simplifies Σ by canceling every clause containing a literal ℓ derived using unit propagation, and shortens every clause containing a complementary literal $\sim\ell$. For efficiency reasons, such a simplification is not "physically" performed on the CNF (instead the set of literals derived using unit propagation is maintained).

– $\Sigma[\ell \leftarrow \Phi]$ denotes the CNF formula obtained by first replacing in the CNF formula Σ every occurrence of ℓ (resp. $\sim\ell$) by Φ (resp. $\neg\Phi$), then turning the resulting formula into an equivalent CNF one by removing every connective different from $\neg$, $\wedge$, $\vee$, using distribution laws, removing in every clause every occurrence of a multi-occurrent literal but one, and finally removing the valid clauses which could be generated.

Example 1 As a matter of illustration, let Σ be the CNF formula consisting of the following clauses:

$\neg a$,
$\neg a \vee b$,
$a \vee c$,
$\neg c \vee d$,
$\neg c \vee e \vee f$,
$f \vee \neg g$.

Here $\mathtt{bcp}(\Sigma) = \{\neg a, c, d\}$. The side effect of the unit propagation is that the clauses $\neg a$, $\neg a \vee b$, $a \vee c$, $\neg c \vee d$ are cancelled, while $\neg c \vee e \vee f$ is simplified as $e \vee f$.

On the other hand, let Ψ be the CNF formula consisting of the three clauses below on the left, and let $\Phi = \neg a \wedge d \wedge \neg e$. $\Psi[c \leftarrow \Phi]$ coincides with Ψ except that the clauses hereafter on the left are replaced by the clauses on the right:

$a \vee c$,
$\neg c \vee d$,
$\neg c \vee e \vee f$,
$a \vee d$,
$a \vee \neg e$,
$a \vee \neg d \vee e \vee f$.

Going into more details, the replacement of c by $\neg a \wedge d \wedge \neg e$ in $a \vee c$ leads to the generation of three clauses: $a \vee \neg a$, which is removed since it is valid, $a \vee d$, and $a \vee \neg e$; the replacement of c by $\neg a \wedge d \wedge \neg e$ in $\neg c \vee d$ leads to the generation of $a \vee \neg d \vee e \vee d$, which is removed since it is valid; the replacement of c by $\neg a \wedge d \wedge \neg e$ in $\neg c \vee e \vee f$ leads to the generation of $a \vee \neg d \vee e \vee e \vee f$, which is simplified as $a \vee \neg d \vee e \vee f$.

3 Preprocessings for Model Counting

We have studied and evaluated the following elementary preprocessing techniques for model counting: backbone detection, occurrence reduction, vivification, as well as literal equivalence, AND, and XOR gate identification and replacement. We also investigated two combinations of elementary preprocessings, eq and $\#eq$.

Each elementary preprocessing technique under consideration p preserves the number of models of the input CNF formula Σ, but some of them are even equivalence-preserving. Beyond equivalence-preservation, other properties of interest can be considered for p, especially the fact that p is *confluent* or not (which expresses whether or not $p(\Sigma)$ is sensitive w.r.t. the way clauses and literals in them are listed in Σ) and the fact that p is *projective* or not (which amounts to determining whether $p(p(\Sigma)) = p(\Sigma)$, and is useful to decide whether iterating p can prove useful or not).

In order to determine how much each technique p leads to "simplify" the input CNF formula Σ, we considered three measures of the "simplicity" of Σ: $\#var(\Sigma)$,

the number of variables of Σ, $\#lit(\Sigma)$, the number of literals occurring in Σ, and $tw(\Sigma)$, the treewidth of the primal graph associated with Σ. We thus compared for each p $\#var(\Sigma)$ with $\#var(p(\Sigma))$, $\#lit(\Sigma)$ with $\#lit(p(\Sigma))$, and $tw(\Sigma)$ with $tw(p(\Sigma))$. Empirically, the results are presented on scatter plots, where each point corresponds to an instance Σ, its x-coordinate corresponds to the value measured on Σ, while its y-coordinate corresponds to the value measured on $p(\Sigma)$. The scales used for both coordinates are logarithmic ones.

The rationale for considering the number of variables and the number of literals of a CNF formula Σ (i.e., the size of the input up to a constant factor) is that the complexity of counting the number of models of Σ depends on both of them: the smaller the better. However, the nature of the instance also has a tremendous impact on the complexity of the algorithm: small instances can prove much more difficult to solve than much bigger ones. Stated otherwise, preprocessing does not mean compressing. Clearly, it would be inadequate to restrict the family of admissible preprocessing techniques to those for which no space increase is guaranteed. Indeed, adding some redundant information can be a way to enhance the instance solving since the pieces of information which are added can lead to an improved propagation power of the solver (see e.g., [11,36]). Especially, some approaches to knowledge compilation consists in adding redundant clauses to the input CNF formula in order to make it unit-refutation complete [19,9,8]. This is the reason why we also considered the treewidth measure. Treewidth is a structural parameter which is widely used in the complexity analysis of graph algorithms. Many of them require exponential time only of the treewidth of the input graph (and not on the size of the graph). Especially, counting the number of models of a CNF formula Σ is fixed-parameter tractable when the parameter is $tw(\Sigma)$ [49]. A significant decrease of $\#var(\Sigma)$, $\#lit(\Sigma)$ or $tw(\Sigma)$ may explain an improved performance of the subsequent model counting process.

For each p, we considered both the best and the worst case scenarios; we also evaluated the impact of p in practice by considering 182 CNF instances 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)

All the instances come from the SAT LIBrary (available at `www.cs.ubc.ca/~hoos/SATLIB/index-ubc.html`). The experiments have been conducted on Intel Xeon E5-2643 (3.30GHz) processors with 32 GiB RAM on Linux CentOS. A time-out of 1h and a memory-out of 7.6 GiB has been considered for each instance.

Since computing the treewidth of a graph is a NP-hard problem, we computed only a upper bound of $tw(\Sigma)$ using `QuickBB` (available at `www.hlt.utdallas.edu/~vgogate/quickbb.html`) equipped with the `min fill` heuristic and

for an allocated time of 1h. When no preprocessing has been performed, `QuickBB` terminated with a memory-out much of the time (for 102 instances over 182, including many instances from the Bayesian networks family and from the BMC family, and all instances from the scheduling data set and of the Qif data set). In such cases, no approximation of the treewidth was returned. This explains why the number of dots in the scatter plots reporting the tw value is significantly lower than the number of dots in the scatter plots related to $\#var$ or to $\#lit$ (which are easy to be computed). Note also that `QuickBB` succeeded in computing the exact value of the treewidth, only for 12 instances.

3.1 Equivalence-Preserving Preprocessings

3.1.1 Backbone Identification

The backbone [41] 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 backbone identification (cf. Algorithm 1) is to make the backbone B of the input `CNF` formula Σ explicit, to conjoin it to Σ, and to use `bcp` on the resulting set of clauses. The search of literals participating to the backbone is limited to the literals ℓ satisfied by a model $\mathcal{I}$ of Σ computed first. If $\Sigma \wedge \sim\ell$ is contradictory, then every model of Σ must satisfy ℓ, and ℓ must belong to B. Otherwise $\Sigma \wedge \sim\ell$ has a model $\mathcal{I}'$ and only the literals satisfied by both $\mathcal{I}$ and $\mathcal{I}'$ may belong to B (thus, the subsequent search can be reduced).

Backbone identification preserves equivalence, and (obvioulsy) it is confluent and projective. However, it may require exponential time (since we use a complete SAT solver `solve` for achieving the satisfiability tests). In our implementation, `solve` exploits *assumptions*; especially clauses which are learnt at each call to `solve` are kept for the subsequent calls; this has a significant impact on the efficiency of the whole process [3].

Algorithm 1: `backboneSimpl`

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

1 $\mathcal{B} \leftarrow \emptyset$;
2 $\mathcal{I} \leftarrow$ `solve`(Σ);
3 **while** $\exists \ell \in \mathcal{I}$ *s.t.* $\ell \notin \mathcal{B}$ **do**
4 $\quad \mathcal{I}' \leftarrow$ `solve`$(\Sigma \cup \{\sim\ell\})$;
5 $\quad$ **if** $\mathcal{I}' = \emptyset$ **then** $\mathcal{B} \leftarrow \mathcal{B} \cup \{\ell\}$ **else** $\mathcal{I} \leftarrow \mathcal{I} \cap \mathcal{I}'$;
6 **return** `bcp`$(\Sigma \cup \mathcal{B})$

Example 2 Let Σ be the `CNF` formula consisting of the following clauses:

$a \vee b$, $c \vee d$,
$\neg a \vee b$, $\neg c \vee e \vee f$,
$\neg b \vee c$, $f \vee \neg g$.

The backbone of Σ is equal to $B = \{b, c\}$.
$\texttt{backboneSimpl}(\Sigma)$ consists of the following clauses:

b, $e \vee f$,
c, $f \vee \neg g$.

Clearly enough, we have $\#var(\texttt{backboneSimpl}(\Sigma)) \leq \#var(\Sigma)$: no variable is added but some variables can be easily removed due to the Boolean constraint propagation which is performed once the literals of the backbone have been found.

Furthermore, we also have that:

$$\#lit(\texttt{backboneSimpl}(\Sigma)) \leq \#lit(\Sigma).$$

Since no new clauses are added to Σ, except unit ones, the primal graph of the CNF formula $\texttt{backboneSimpl}(\Sigma)$ is a partial graph of the primal graph of Σ. Since the treewidth of any partial graph of a given graph is always lower than or equal to the treewidth of the graph [47], we get that:

$$tw(\texttt{backboneSimpl}(\Sigma)) \leq tw(\Sigma).$$

Interestingly, the reduction achieved by considering backboneSimpl can be arbitrarily large, whatever the considered measure (among the three ones we considered). This can be easily observed by considering the CNF formula $\Sigma = x_0 \wedge (x_0 \vee \ldots \vee x_n)$. Indeed, backboneSimpl leads to a CNF formula $\texttt{backboneSimpl}(\Sigma)$ which is equal to x_0 (thus independent of n).

Empirically, we obtained the results reported on Figures 1, 2, and 3.

One can observe on Figures 1, 2, and 3 that backboneSimpl may lead in practice to significant reductions of both the $\#var$ value, the $\#lit$ value and the tw value of the instance. The backboneSimpl preprocessing looks as particularly useful for the BMC data set and the planning data set, while of limited impact for the Bayesian networks data set, the configuration data set and the circuit dataset. Note also that using the backboneSimpl preprocessing improves both the number of instances for which QuickBB succeeded in computing a tw value (it terminated with a memory-out for 45 instances over 182 – remember that it crashed due to a memory-out for 102 instances over 182 when no preprocessing was performed) and the number of instances for which QuickBB succeeded in computing the exact value of the treewidth (33 instances vs. 12 without any preprocessing).

3.1.2 Occurrence Reduction

Occurrence reduction (cf. Algorithm 2) is a simple procedure we have developed for removing some literals in the input CNF formula Σ via the replacement of some

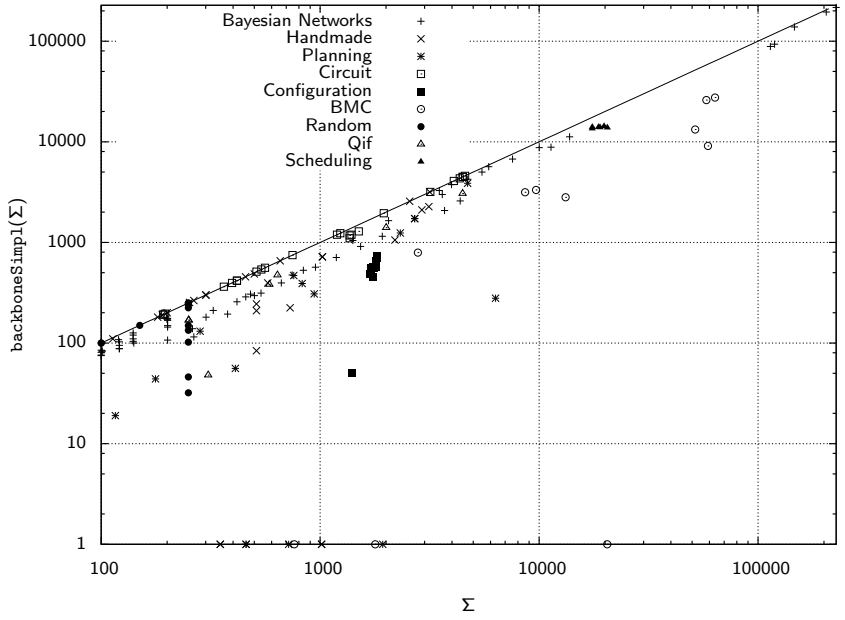

Fig. 1 Comparing $\#var(\Sigma)$ with $\#var(\texttt{backboneSimpl}(\Sigma))$.

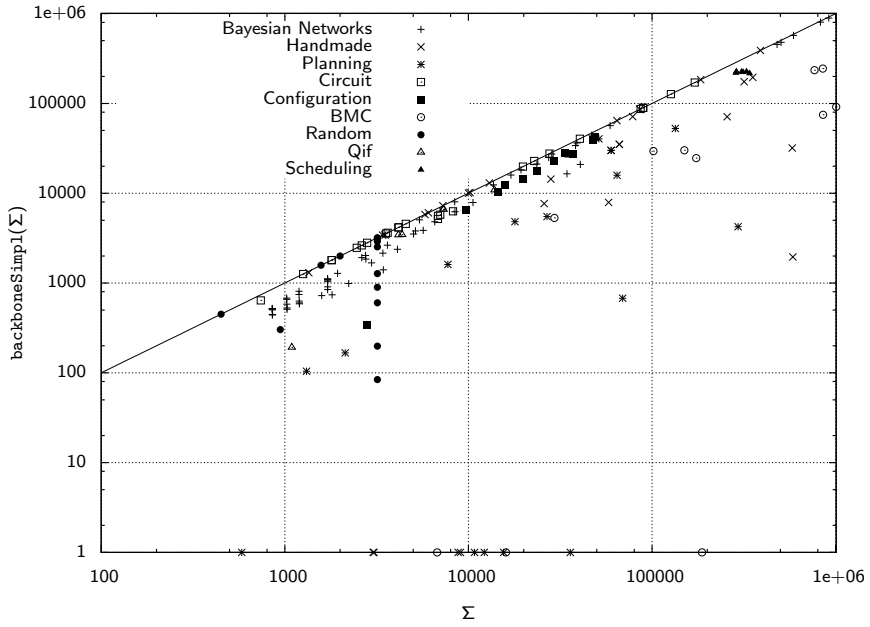

Fig. 2 Comparing $\#lit(\Sigma)$ with $\#lit(\texttt{backboneSimpl}(\Sigma))$.

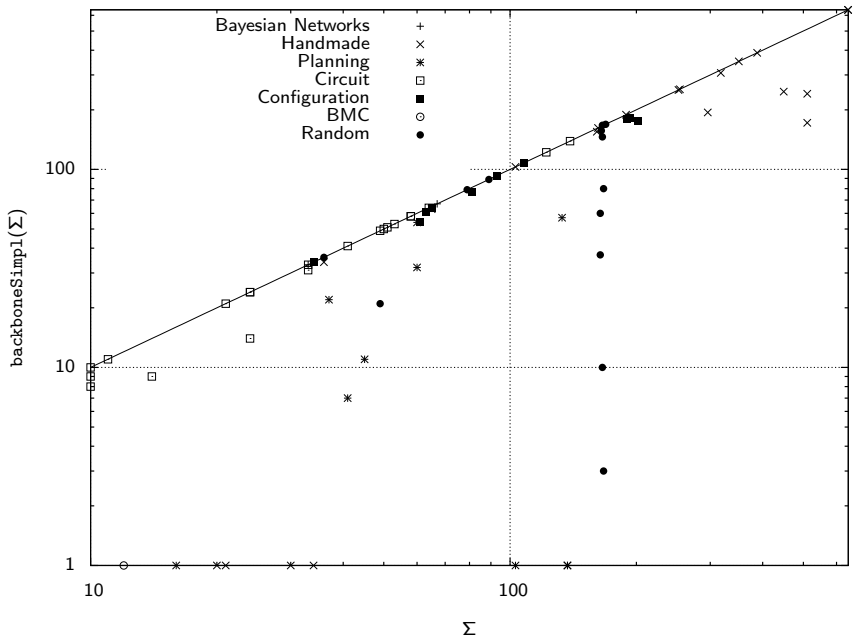

Fig. 3 Comparing $tw(\Sigma)$ with $tw(\texttt{backboneSimpl}(\Sigma))$.

clauses $\alpha = \ell_1 \vee \ldots \vee \ell_j \vee \ell_{j+1}$ by subsuming ones $\alpha \setminus \{\ell_{j+1}\}$. 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. Again, bcp is used as an incomplete yet efficient method to solve the entailment problem. At start, the literals of Σ are sorted by considering their number of occurrences (the greatest first) and put in a list $\mathcal{L}$.

While it is equivalence-preserving, occurrence reduction neither is confluent nor is projective. Especially, the reduction of previous clauses has an impact on the propagation power of bcp, hence on the opportunity to reduce clauses occurring next. For the same reason, it is not projective (when the clauses "occurring next" are the first clauses of Σ considered during the next round when occurrence reduction is repeated).

Occurrence reduction has a worst-case time complexity cubic in the input size. Occurrence reduction can also be viewed as a light form of vivification that will be presented just after (the objective is just to remove literals and not clauses). Compared to vivification, the rationale for keeping some redundant clauses is that this may lead to an increased inferential power w.r.t. unit propagation.

Example 3 Let Σ be the CNF formula consisting of the following clauses:

Algorithm 2: `occurrenceSimpl`

input : a CNF formula Σ
output : a CNF formula equivalent to Σ
1 $\mathcal{L} \leftarrow sort(Lit(\Sigma))$;
2 **foreach** $\ell \in \mathcal{L}$ **do**
3 **foreach** $\alpha \in \Sigma$ *s.t.* $\ell \in \alpha$ **do**
4 **if** $\emptyset \in \mathtt{bcp}(\Sigma \cup \{\ell\} \cup \{\sim(\alpha \setminus \{\ell\})\})$ **then** $\Sigma \leftarrow (\Sigma \setminus \{\alpha\}) \cup \{\alpha \setminus \{\ell\}\}$;
5 **return** Σ

$a \vee f$, $b \vee d \vee e$, $b \vee d \vee \neg e$,
$a \vee b \vee c$, $c \vee \neg d \vee e$, $c \vee \neg d \vee \neg e$.

Among the possible lists $\mathcal{L}$ obtained at line 1 are $\mathcal{L}_1 = (b, c, e, \neg e, d, \neg d, a, f, \neg a, \neg b, \neg c)$ and $\mathcal{L}_2 = (b, c, a, e, \neg e, d, \neg d, f, \neg a, \neg b, \neg c)$. If $\mathcal{L}_1$ is chosen, then `occurrenceSimpl`(Σ) consists of the following clauses:

$a \vee f$, $b \vee d$, $b \vee d$,
$b \vee c$, $c \vee \neg d$, $c \vee \neg d$.

The effect of `occurrenceSimpl` is to reduce $a \vee b \vee c$ to $b \vee c$, and to eliminate every occurrence of the variable e. We can observe that `occurrenceSimpl` neither is confluent, nor is projective. Indeed, if $\mathcal{L}_2$ was chosen, then `occurrenceSimpl`(Σ) would consist of the following clauses:

$a \vee f$, $b \vee d$, $b \vee d$,
$a \vee b \vee c$, $c \vee \neg d$, $c \vee \neg d$.

This shows that `occurrenceSimpl` is not confluent. For the literal ordering $\mathcal{L}_2$, the clause $a \vee b \vee c$ has not been simplified into $b \vee c$. However, if `occurrenceSimpl` was applied once more while considering the list $\mathcal{L}_2$, we would have obtained `occurrenceSimpl`(`occurrenceSimpl`(Σ)):

$a \vee f$, $b \vee d$, $b \vee d$,
$b \vee c$, $c \vee \neg d$, $c \vee \neg d$.

showing that `occurrenceSimpl` is not projective.

Since `occurrenceSimpl` does not generate any new variable, we have that:

$$\#var(\mathtt{occurrenceSimpl}(\Sigma)) \leq \#var(\Sigma).$$

Similarly, since `occurrenceSimpl` consists in removing some literals in Σ, it cannot lead to increase the size of Σ, thus we have:

$$\#lit(\mathtt{occurrenceSimpl}(\Sigma)) \leq \#lit(\Sigma).$$

Finally, since no new clauses are added to Σ, except unit ones, the primal graph of $\texttt{occurrenceSimpl}(\Sigma)$ is a partial graph of the primal graph of Σ, so we get that:

$$tw(\texttt{occurrenceSimpl}(\Sigma)) \leq tw(\Sigma).$$

The previous example considered for `backboneSimpl` can also be considered here to show that the `occurrenceSimpl` preprocessing may lead to an arbitrarily large reduction of the $\#var$ value, the $\#lit$ value, and the tw value of the instance.

Empirically, we obtained the results reported on Figures 4, 5, and 6.

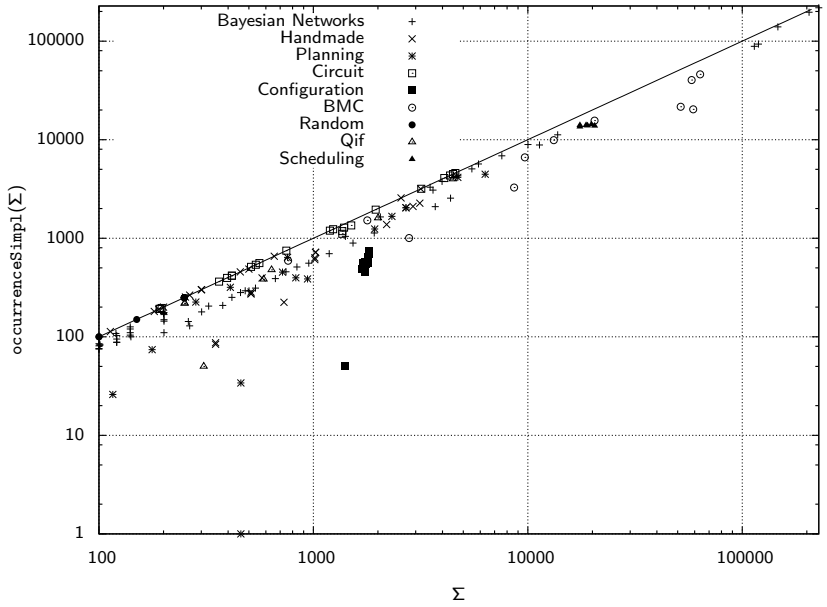

Fig. 4 Comparing $\#var(\Sigma)$ with $\#var(\texttt{occurrenceSimpl}(\Sigma))$.

One can observe on Figures 4, 5, and 6 that `occurrenceSimpl` may lead in practice to valuable reductions of both the $\#var$ value, the $\#lit$ value and the tw value of the instance. As it was the case for `backboneSimpl`, the impact on the tw value is often higher than the impact on the $\#lit$ value, which is itself higher than the impact on the $\#var$ value. The `occurrenceSimpl` preprocessing looks as useful for the planning data set. The few dots slightly above the diagonal on Figure 6 can be viewed, so to say, as "empirical errors"; they correspond to instances for which applying the `occurrenceSimpl` preprocessing leads to degrade the value of the estimation of the treewidth, as computed by `QuickBB` (such dots would not exist if the exact value of the treewidth was computed). Finally, note that using `occurrenceSimpl` improves significantly both the number of instances for which `QuickBB` succeeded in computing a tw value (it terminated with a memory-out for 49 instances over 182) and

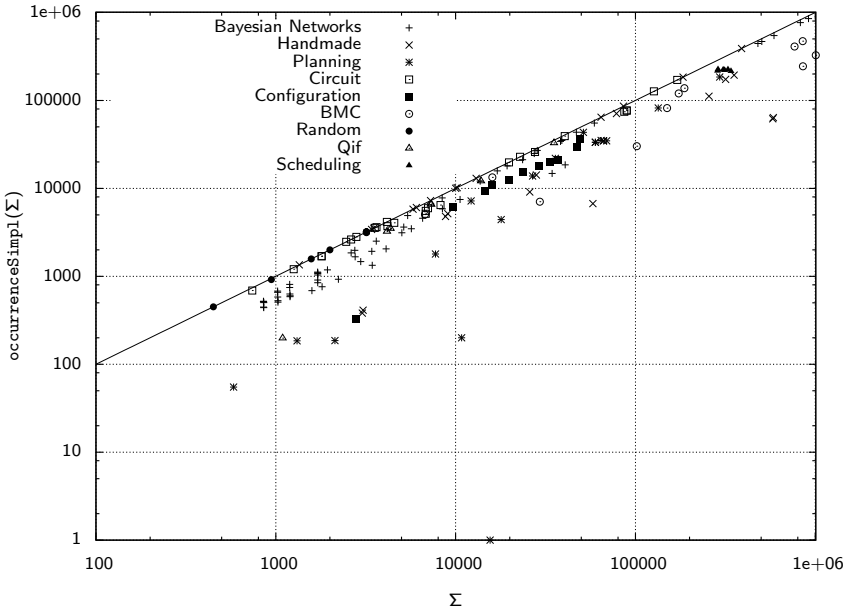

Fig. 5 Comparing $\#lit(\Sigma)$ with $\#lit(\texttt{occurrenceSimpl}(\Sigma))$.

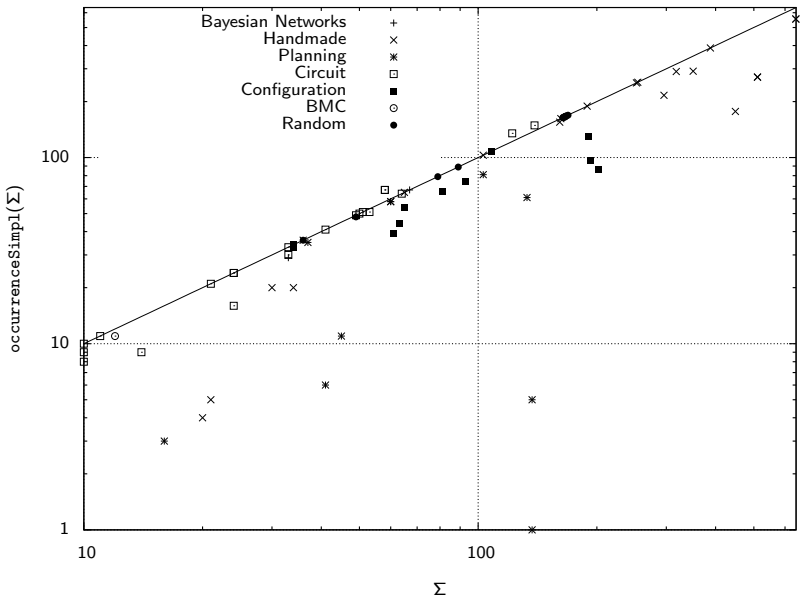

Fig. 6 Comparing $tw(\Sigma)$ with $tw(\texttt{occurrenceSimpl}(\Sigma))$.

the number of instances (24) for which `QuickBB` succeeded in computing the exact value of the treewidth.

Algorithm 3: `vivificationSimpl`

```
   input  : a CNF formula Σ
   output : a CNF formula equivalent to Σ
 1 foreach α ∈ Σ do
 2     Σ←Σ \ {α};
 3     α'←⊥;
 4     I←bcp(Σ);
 5     while ∃ℓ ∈ α s.t. ∼ℓ ∉ I and α' ≠ ⊤ do
 6         α'←α' ∨ ℓ;
 7         I←bcp(Σ ∪ {∼α'});
 8         if ∅ ∈ I then α'←⊤;
 9     Σ←Σ ∪ {α'};
10 return Σ
```

3.1.3 Vivification

Vivification (cf. Algorithm 3) [46] is a preprocessing technique which aims at reducing the given CNF formula Σ, i.e., to remove some clauses and some literals in Σ while preserving equivalence. Its time complexity is in the worst case cubic in the input size.

Basically, 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. Thus, for each clause α of Σ the literals ℓ_{j+1} of α are successively considered and the question is to determine whether they should be added or not to the current subclause $\alpha' = \ell_1 \vee \ldots \vee \ell_j$ of α (initialized as the empty clause at line 3). 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. This is ensured by the assignment $\alpha' \leftarrow \top$ at line 8 given the test $\alpha' \neq \top$ at line 5). 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 (this motivates the test $\sim\ell \notin \mathcal{I}$ at line 5 – remind that $\mathcal{I}$ is at any step the set of literals which can be obtained from $(\Sigma \setminus \{\alpha\}) \cup \{\sim\alpha'\}$ using unit propagation).

The clauses α of Σ are considered sequentially, i.e., the first clause of Σ is considered first, and so on. If the backbone identification preprocessing has been performed first, literals are handled (line 5) based on the VSIDS (*Variable State Independent, Decaying Sum*) [42] activities (the most active ones first) of the corresponding variables, as computed by `solve` (line 2 of `backboneSimpl`); otherwise, they are considered w.r.t. the lexicographic ordering. Note that the case $\ell \in \mathcal{I}$ does not need to be considered explicitly at line 5 of `vivificationSimpl`. Indeed, if ℓ belongs to $\mathcal{I}$, then since ℓ belongs to α' (due to line 6), bcp$(\Sigma \cup \{\sim\alpha'\})$ generates $\{\emptyset\}$ at line 7 (a contradiction is detected); this implies that the clause α at hand is removed at line 8 (it is replaced by the clause α' at line 9, and α' has been set to $\top$ in this case).

Example 4 Let Σ be the CNF formula consisting of the following clauses:

$a \vee b \vee c \vee d$,
$a \vee b \vee c$,
$a \vee \neg d$.

Suppose that the VSIDS scores of the four variables are identical, and assume that the variables are processed w.r.t. the ordering $d < c < b < a$. The CNF formula $\texttt{vivificationSimpl}(\Sigma)$ consists of the following clauses:

$a \vee b \vee c$,
$a \vee \neg d$.

The effect of `vivificationSimpl` on Σ is to eliminate the first clause of it, namely $a \vee b \vee c \vee d$. Suppose now that the clauses of Σ were considered in the following ordering (where the first two clauses of Σ have been switched):

$a \vee b \vee c$,
$a \vee b \vee c \vee d$,
$a \vee \neg d$.

In this case, $\texttt{vivificationSimpl}(\Sigma)$ would consist of the following clauses:

$a \vee b \vee c \vee d$,
$a \vee \neg d$.

This shows that `vivificationSimpl` is not confluent. If `vivificationSimpl` was applied once more on the resulting clauses while considering this time the variable ordering $a < d < c < b$ (which is possible since the VSIDS scores of the four variables are identical), then one would get $\texttt{vivificationSimpl}(\texttt{vivificationSimpl}(\Sigma))$:

$a \vee b \vee c$,
$a \vee \neg d$.

showing that `vivificationSimpl` is not projective.

The same observations as those made for `occurrenceSimpl` can also be done for `vivificationSimpl`, enabling to conclude that:

$$\#var(\texttt{vivificationSimpl}(\Sigma)) \leq \#var(\Sigma),$$

$$\#lit(\texttt{vivificationSimpl}(\Sigma)) \leq \#lit(\Sigma),$$

$$tw(\texttt{occurrenceSimpl}(\Sigma)) \leq tw(\Sigma).$$

The previous example considered for `backboneSimpl` can also be considered here to show that the `vivificationSimpl` preprocessing may lead to an arbitrarily large reduction of the $\#var$ value, the $\#lit$ value, and the tw value of the instance.

Empirically, we obtained the results reported at Figures 7, 8, and 9.

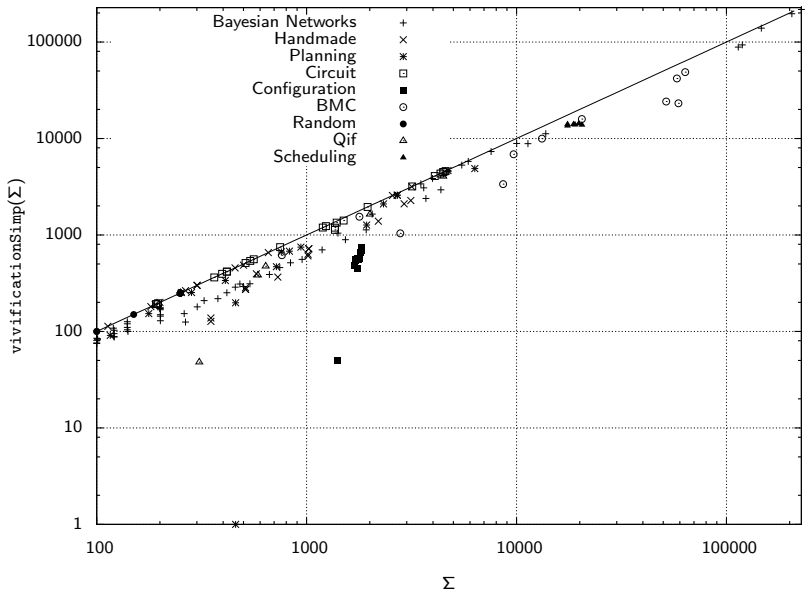

Fig. 7 Comparing $\#var(\Sigma)$ with $\#var(\texttt{vivificationSimpl}(\Sigma))$.

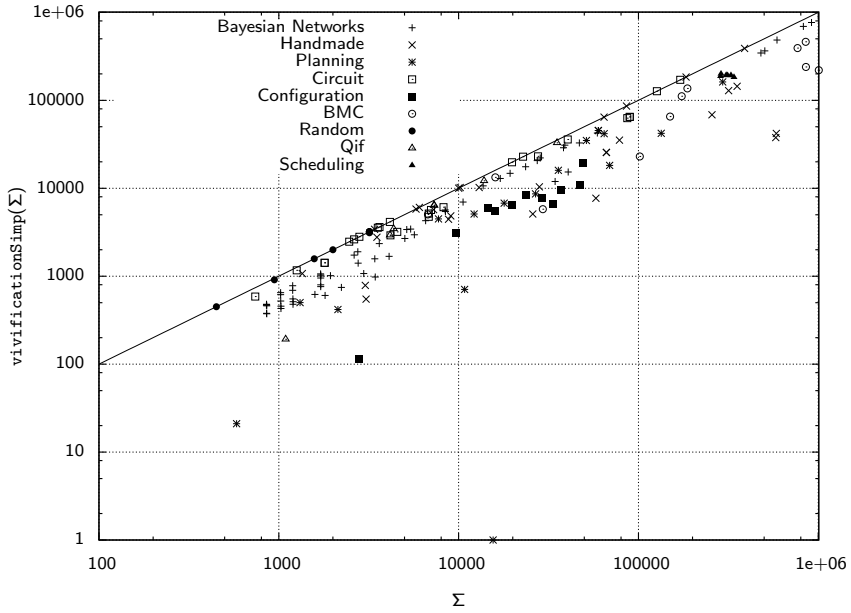

Fig. 8 Comparing $\#lit(\Sigma)$ with $\#lit(\texttt{vivificationSimpl}(\Sigma))$.

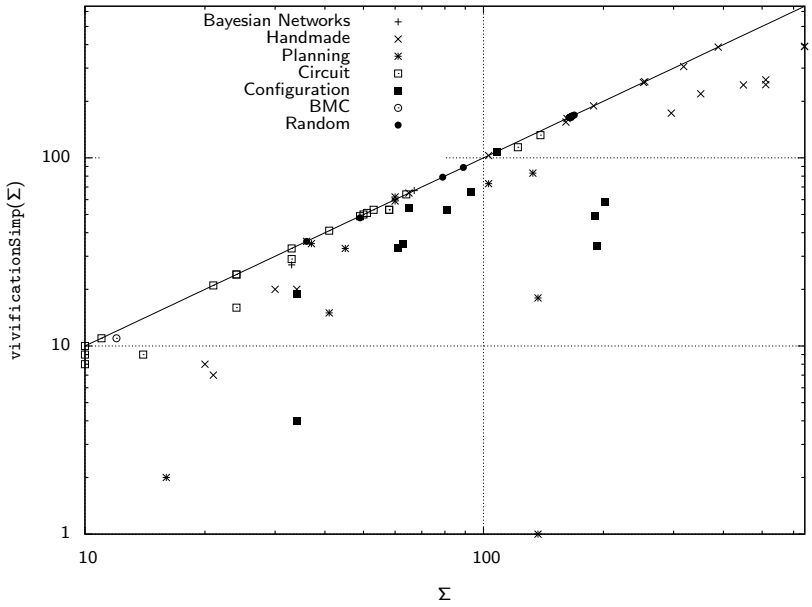

Fig. 9 Comparing $tw(\Sigma)$ with $tw(\texttt{vivificationSimpl}(\Sigma))$.

Figures 7, 8, and 9 show that the `vivificationSimpl` preprocessing may lead in practice to valuable reductions of both the $\#var$ value, the $\#lit$ value and the tw value of the instance. The improvements are quite similar to the ones achieved by `occurrenceSimpl`. The `vivificationSimpl` preprocessing looks as useful for the planning data set, and for the configuration data set. Note also that using the `vivificationSimpl` preprocessing improves both the number of instances for which `QuickBB` succeeded in computing a tw value (it terminated with a memory-out for 48 instances over 182) and the number of instances (29) for which `QuickBB` succeeded in computing the exact value of the treewidth.

3.2 Literal Equivalences and Gates Detection and Replacement

Literal equivalences and gates detection and replacement are preprocessing techniques which do not preserve equivalence but only the number of models of the input formula. The detection of equivalent literals was used for preprocessing in [5], while AND gates and XOR gates detection and replacement have been exploited in [44] and in [21].

The correctness of those preprocessing techniques relies on the following principle: given two propositional formulae Σ and Φ and a literal ℓ, if $\Sigma \models \ell \leftrightarrow \Phi$ holds, then $\Sigma[\ell \leftarrow \Phi]$ has the same number of models as Σ. Implementing it requires first to detect a logical consequence $\ell \leftrightarrow \Phi$ of Σ, then to perform the replacement $\Sigma[\ell \leftarrow \Phi]$ (and in our case, turning the resulting formula into an equivalent `CNF`). In

our approach, replacement is performed only if it is not too space inefficient (this is reminiscent to NIVER [54], which allows for applying the variable elimination rule on a formula if this does not lead to increase its size). This is guaranteed in the literal equivalence case, i.e., when Φ is a literal but not in the remaining cases in general – AND gate when Φ is a term (or dually a clause) and XOR gate when Φ is a XOR clause (or dually a chain of equivalences).

3.2.1 Literal Equivalences

Literal equivalence detection and replacement is a preprocessing technique at work in SAT solvers [21]. Algorithm 4 presents how this preprocessing is performed.

Some previous techniques for literal equivalence detection and replacement are based on pattern checking, searching for binary clauses in the input CNF formula Σ, encoding literal equivalences, and more generally looking at strongly connected components of the binary implication graph of the input CNF formula Σ [24]. Contrastingly, in our approach, bcp is used for detecting equivalences between literals. For each literal ℓ, all the literals ℓ' which can be found equivalent to ℓ using bcp are replaced by ℓ in Σ. Interestingly, taking advantage of bcp makes it more efficient (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).

In the worst case, the time complexity of equivSimpl is cubic in the input size.[3]

Algorithm 4: equivSimpl

```
   input  : a CNF formula Σ
   output : a CNF formula Φ such that ‖Φ‖ = ‖Σ‖
 1 Φ←Σ;
 2 Unmark all variables of Φ;
 3 while ∃ℓ ∈ Lit(Φ) s.t. var(ℓ) is not marked do
       // detection
 4     mark var(ℓ);
 5     P_ℓ←BCP(Φ ∪ {ℓ});
 6     N_ℓ←BCP(Φ ∪ {∼ℓ});
 7     Γ←{ℓ ↔ ℓ' | ℓ' ≠ ℓ and ℓ' ∈ P_ℓ and ∼ℓ' ∈ N_ℓ};
       // replacement
 8     foreach ℓ ↔ ℓ' ∈ Γ do
 9         replace ℓ by ℓ' in Φ;
10 return Φ
```

In equivSimpl, the literals of the input CNF formula Σ are processed w.r.t. the lexicographic ordering of the corresponding variables. Since the chosen representative of each literal equivalence class depends on the syntactic presentation of the input CNF Σ and since the replacement step may lead to reduce some clauses, equivSimpl neither is confluent nor is projective.

[3] Literals are degenerate AND gates and degenerate XOR gates; however equivSimpl may detect equivalences that would not be detected by ANDgateSimpl or by XORgateSimpl; this explains why equivSimpl is used.

Example 5 Let Σ be the CNF formula consisting of the following clauses:

$a \vee b \vee c \vee \neg d$, $\neg a \vee b$, $e \vee \neg g$,
$\neg a \vee \neg b \vee \neg c \vee d$, $a \vee \neg b$, $\neg e \vee \neg h$.
$a \vee b \vee \neg c$, $\neg e \vee \neg f \vee h$,
$\neg a \vee \neg b \vee c$, $e \vee \neg f \vee g$,

Assume that the variables of Σ are considered in the following ordering: $a < b < c < d < e < f < g < h$. Then equivSimpl($\Sigma$) consists of:

$\neg f \vee \neg g$,
$f \vee \neg h$.

Indeed, at the first step, when a is considered, the equivalence $a \leftrightarrow b$ is detected and a is replaced by b. Then, when b is considered, the equivalence $b \leftrightarrow c$ is detected and b is replaced by c. When c is considered, the equivalence $c \leftrightarrow d$ is detected and c is replaced by d. At this step, the first six clauses of Σ do not belong any longer to it, because the successive replacements have turned them into valid clauses. Now, when e is considered, we get $\mathcal{P}_e = \{e, \neg h, \neg f\}$ and $\mathcal{N}_e = \{\neg e, \neg g, f\}$. Since $\neg f \in \mathcal{P}_e$ and $f \in \mathcal{N}_e$, the equivalence $e \leftrightarrow \neg f$ has been found. Replacing e by $\neg f$ in the current set of clauses leads to the two clauses $\neg f \vee \neg g$, $f \vee \neg h$, as expected. Observe that the equivalence $e \leftrightarrow \neg f$ is detected thanks to bcp (unlike what happens for $a \leftrightarrow b$, the two clauses $e \vee f$ and $\neg e \vee \neg f$ stating this equivalence are not explicitly present in Σ at start).

Assume now that the variables of Σ were considered w.r.t. the following ordering: $e < f < g < h < d < c < b < a$. Then equivSimpl would have consisted of:

$c \vee \neg d$, $\neg f \vee \neg g$,
$\neg c \vee d$, $f \vee \neg h$.

Indeed, nothing changes for the clauses of Σ built up from $\{e, f, g, h\}$ (since they are disconnected to the other clauses of Σ). Then, when d and c are successively considered, no equivalences are detected. When b is considered, the equivalence $b \leftrightarrow a$ is detected and b is replaced by a. Finally, when a is considered, the equivalence $a \leftrightarrow c$ is detected and a is replaced by c. This shows that equivSimpl is not confluent. Furthermore, equivSimpl is not projective since if equivSimpl is applied once more to the resulting set of clauses, then the equivalence $d \leftrightarrow c$ will be detected and d will be replaced by c, leading to the following set of clauses:

$\neg f \vee \neg g$,
$f \vee \neg h$.

With equivSimpl, each time a literal equivalence is detected, one variable is eliminated, which shows that:

$$\#var(\texttt{equivSimpl}(\Sigma)) \leq \#var(\Sigma).$$

Similarly, it is obvious that the replacement of any literal by a literal cannot increase the total number of literals. A decrease may easily occur because in the generated clauses every occurrence of a multi-occurrent literal (but one) is removed, and valid clauses are eliminated. Thus we have:

$$\#lit(\texttt{equivSimpl}(\Sigma)) \leq \#lit(\Sigma).$$

Contrastingly, applying `equivSimpl` does not ensure that the tw score of the instance Σ evolves in a monotone way. Thus, an arbitrarily large tw decrease may happen in some cases. For instance, consider $\Sigma = \bigwedge_{i=0}^{n-1}(\neg x_i \vee x_{i+1}) \wedge (\neg x_n \vee x_0) \wedge \bigvee_{i=0}^{n} x_i$. Applied to it, `equivSimpl` leads to a CNF formula $\texttt{equivSimpl}(\Sigma)$ such that the cardinality of $Var(\texttt{equivSimpl}(\Sigma))$ is 1 (n variables are eliminated). Accordingly, this example shows that an arbitrarily large decrease of each of the three measures ($\#var$, $\#lit$, tw) may result from the application of `equivSimpl`.
However, a treewidth increase may also happen in some cases. Here is an example showing it; let $\Sigma = (\neg x_0 \vee x_1) \wedge (\neg x_1 \vee x_0) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_4) \wedge (\neg x_4 \vee x_3) \wedge (\neg x_3 \vee x_1) \wedge (\neg x_3 \vee x_5) \wedge (\neg x_5 \vee x_3)$.

Figure 10 gives a representation of the primal graph of Σ. We have $tw(\Sigma) = 2$.

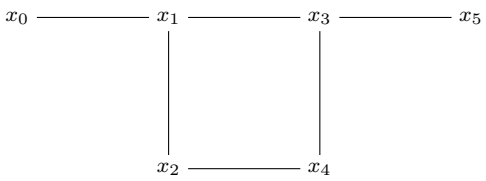

Fig. 10 The primal graph of Σ.

Suppose now that the equivalences $x_0 \leftrightarrow x_4$ and $x_5 \leftrightarrow x_2$ are detected. If x_0 and x_5 are successively replaced by their definitions, then one obtains the CNF formula $(\Sigma[x_0 \leftarrow x_4])[x_5 \leftarrow x_2] = (\neg x_4 \vee x_1) \wedge (\neg x_1 \vee x_4) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_4) \wedge (\neg x_4 \vee x_3) \wedge (\neg x_3 \vee x_1) \wedge (\neg x_3 \vee x_2) \wedge (\neg x_2 \vee x_3)$.

Figure 11 gives a representation of the primal graph of $(\Sigma[x_0 \leftarrow x_4])[x_5 \leftarrow x_2]$. We have $tw((\Sigma[x_0 \leftarrow x_4])[x_5 \leftarrow x_2]) = 3$.

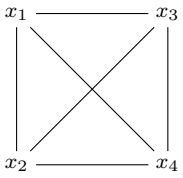

Fig. 11 The primal graph of $(\Sigma[x_0 \leftarrow x_4])[x_5 \leftarrow x_2]$.

It must be noted that when `equivSimpl` is used, each time an equivalence $l \leftrightarrow l'$ is detected, then if there is an edge $\{var(l), var(l')\}$ in the primal graph of Σ (i.e., there is a clause containing the two variables), then the replacement of l by l' (or vice-versa) in Σ leads to a `CNF` formula the primal graph of which has the same treewidth as $tw(\Sigma)$. Indeed, in such a case, the replacement corresponds to an operation of edge contraction, and the resulting graph is thus a minor of the initial graph; in such a case, it is known that the treewidth remains unchanged [47].

Empirically, we obtained the results reported at Figures 12, 13, and 14.

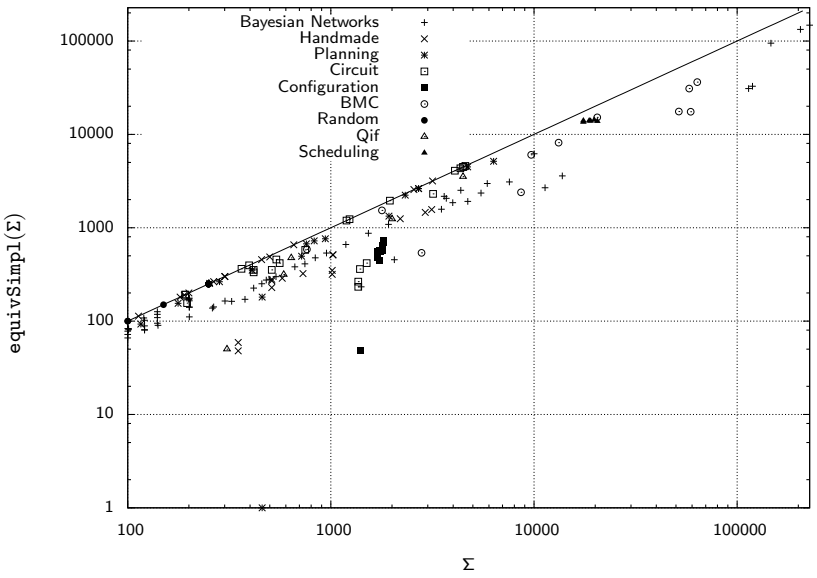

Fig. 12 Comparing $\#var(\Sigma)$ with $\#var(\texttt{equivSimpl}(\Sigma))$.

Figures 12, 13, and 14 show how `equivSimpl` leads to a reduction of both the $\#var$ value, the $\#lit$ value and the tw value of the instance. We can observe that the instances for which huge reductions (i.e., one order of magnitude and above) are obtained are less numerous than the instances for which huge reductions are obtained when the previous, equivalence-preserving preprocessings are considered. Our results have also shown that the `equivSimpl` preprocessing improves both the number of instances for which `QuickBB` succeeded in computing a tw value (it terminated with a memory-out for 46 instances over 182) and the number of instances (19) for which `QuickBB` succeeded in computing the exact value of the treewidth. Finally, while a treewidth increase may result in theory from applying `equivSimpl`, one may observe that it does not occur in practice.

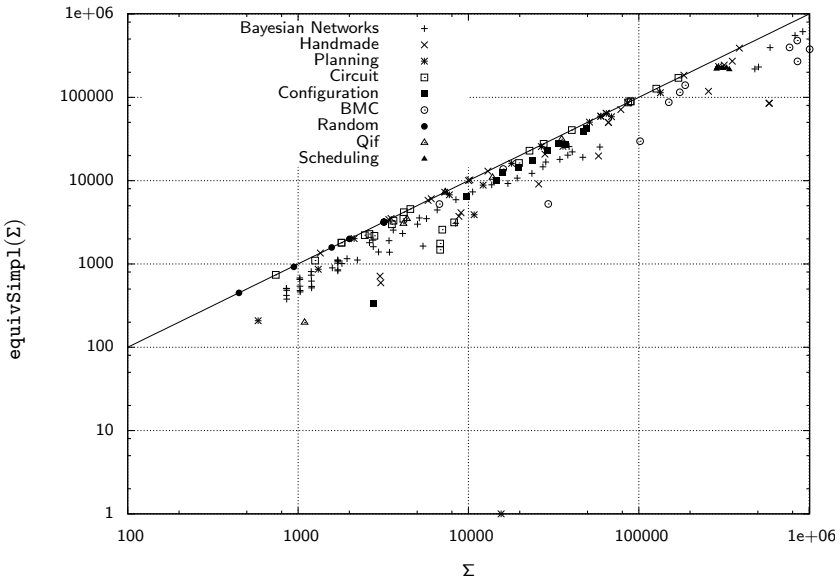

Fig. 13 Comparing $\#lit(\Sigma)$ with $\#lit(\texttt{equivSimpl}(\Sigma))$.

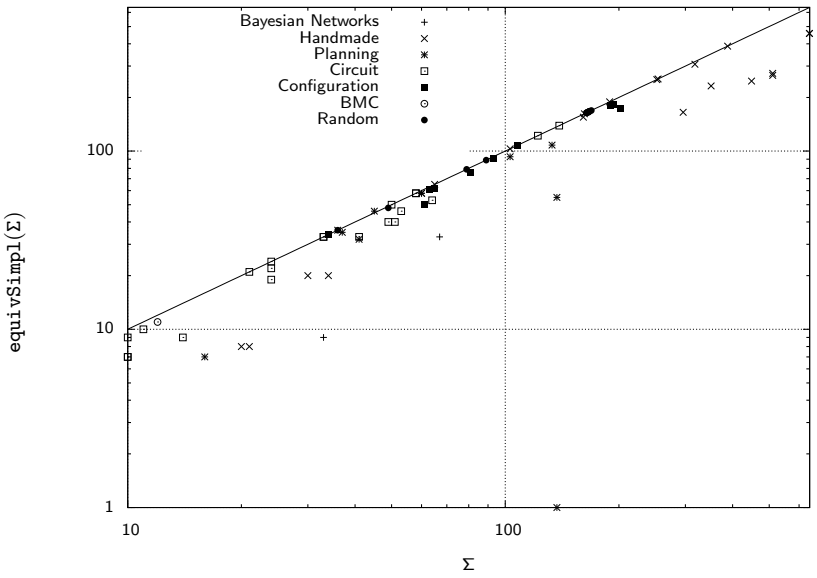

Fig. 14 Comparing $tw(\Sigma)$ with $tw(\texttt{equivSimpl}(\Sigma))$.

3.2.2 AND/OR Gates

First of all, let us recall that, as a trivial consequence of De Morgan's laws, every AND gate (of the form $\ell \leftrightarrow (\ell_1 \wedge \ldots \wedge \ell_k)$) can also be viewed (equivalently) as an

OR gate (of the form $\sim\ell \leftrightarrow (\sim\ell_1 \vee \ldots \vee \sim\ell_k)$). This explains why we treat all such gates (AND gates or OR gates) in a uniform way.

AND/OR gate detection and replacement is presented at Algorithm 5. In the worst case, its time complexity is cubic in the input size. Unlike previous approaches based on pattern matching (i.e., when one looks for clauses encoding an AND gate or an OR gate), our approach takes advantage of `bcp` for detecting such gates, which makes it more efficient (if clauses stating the presence of an AND/OR gate occur in Σ, then `ANDgateSimpl` will find the gate – or a "subsuming" one – but the converse is not true).

Algorithm 5: `ANDgateSimpl`

input : a `CNF` formula Σ
output : a `CNF` formula Φ such that $\|\Phi\| = \|\Sigma\|$
1 $\Phi \leftarrow \Sigma$;
`// detection`
2 $\Gamma \leftarrow \emptyset$;
3 unmark all literals of Φ;
4 **while** $\exists \ell \in Lit(\Phi)$ *s.t.* ℓ *is not marked* **do**
5 mark ℓ;
6 $\mathcal{P}_\ell \leftarrow (BCP(\Phi \cup \{\ell\}) \setminus (BCP(\Phi) \cup \{\ell\})) \cup \{\sim\ell\}$;
7 **if** $\emptyset \in$ `bcp`$(\Phi \cup \mathcal{P}_\ell)$ **then**
8 let $\mathcal{C}_\ell \subseteq \mathcal{P}_\ell$ s.t. $\emptyset \in$ `bcp`$(\Phi \cup \mathcal{C}_\ell)$ and $\sim\ell \in \mathcal{C}_\ell$;
9 $\Gamma \leftarrow \Gamma \cup \{\ell \leftrightarrow \bigwedge_{\ell' \in \mathcal{C}_\ell \setminus \{\sim\ell\}} \ell'\}$;
`// replacement`
10 **while** $\exists \ell \leftrightarrow \beta \in \Gamma$ *st.* $|\beta| < maxA$ *and* $|\Phi[\ell \leftarrow \beta]| \leq |\Phi|$ **do**
11 $\Phi \leftarrow \Phi[\ell \leftarrow \beta]$;
12 $\Gamma \leftarrow \Gamma[\ell \leftarrow \beta]$;
13 $\Gamma \leftarrow \Gamma \setminus \{\ell' \leftrightarrow \zeta \in \Gamma | \ell' \in \zeta\}$
14 **return** Φ

The algorithm starts with a gate detection phase. At line 4, literals ℓ of $Lit(\Sigma)$ are considered w.r.t. any total ordering such that $\sim\ell$ comes just after ℓ. As it is the case with `equivSimpl`, one looks in `ANDgateSimpl` for AND gates defining literals by considering the literals occurring in Σ w.r.t. the lexicographic ordering of the corresponding variables. At line 6, $\mathcal{P}_\ell$ contains $\sim\ell$ and all the literals which can be derived from $\Sigma \wedge \ell$ (using `bcp`), but not from Σ or ℓ taken separately. Hence, after this step, $\mathcal{P}_\ell$ is such that $\Sigma \models \ell \rightarrow (\bigwedge_{\ell' \in \mathcal{P}_\ell \setminus \{\sim\ell\}} \ell')$. The test at line 7 allows for deciding whether an AND gate β with output ℓ exists in Σ. Indeed, if $\Sigma \wedge \bigwedge_{\ell' \in \mathcal{P}_\ell \setminus \{\sim\ell\}} \ell' \wedge \sim\ell$ is unsatisfiable (shown by `bcp`), we have $\Sigma \models (\wedge \bigwedge_{\ell' \in \mathcal{P}_\ell \setminus \{\sim\ell\}} \ell') \rightarrow \ell$. Accordingly, there is an AND gate $\beta = \bigwedge_{\ell' \in \mathcal{P}_\ell \setminus \{\sim\ell\}} \ell'$ with output ℓ in Σ. In our implementation, one tries to minimize the number of variables in this gate (lines 8 and 9) by taking advantage of the implication graph at hand [52,34,42]. Indeed, the unit refutation which has been found from $\Phi \cup \mathcal{P}_\ell$ at line 7 can be viewed as a DAG (the so called implication graph) whose nodes are labelled with literals (alias unit clauses), such that the sources of the graph (i.e., its nodes of in-degree 0) are labelled with literals from $\mathcal{P}_\ell$, there is a node labelled by the empty clause, and each arc (ℓ_i, ℓ_j) in the graph

corresponds to the generation using unit resolution of the literal ℓ_j from ℓ_i using the clause $\sim\ell_i \vee \ell_j$ of Φ. Then the set $\mathcal{C}_\ell$ computed at line 8 is the union of $\{\sim\ell\}$ with the set containing precisely the source literals of the implication graph which can be reached from the node labelled by the empty clause in the graph by considering the arcs back. Note that only one definition per literal ℓ is computed and stored in Γ.

Once all the definitions have been computed, the replacement phase takes place (lines 10 to 13). The definitions $\ell \leftrightarrow \beta$ are sorted by increasing size of β. The replacement of a literal ℓ by the corresponding definition β is performed only if the number of conjuncts in the AND gate β remains "small enough" (i.e., $\leq maxA$ – in our experiments $maxA = 10$). Furthermore, the gate replacement is achieved only if it does not lead to increase the input size. The two conditions at line 10 capture it. The replacement is performed both in the input CNF formula and in the set Γ of definitions (lines 11 and 12). Valid definitions $\ell \leftrightarrow \beta$ are removed from Γ (line 13). Note that a decrease of the instance size may also result from the replacement, because it may produce valid clauses, and such clauses are removed when generated.

Example 6 Let Σ be the CNF formula consisting of the following clauses:

$a \vee b \vee c \vee d$, $\neg a \vee b \vee c \vee \neg d$,
$\neg a \vee \neg b$, $\neg a \vee e$,
$\neg a \vee b \vee \neg c$, $a \vee f$.

Consider the literal ordering: $a < \neg a < b < \neg b < c < \neg c < d < \neg d < e < \neg e < f < \neg f$. $\texttt{ANDgateSimpl}(\Sigma)$ consists of:

$b \vee c \vee d \vee e$, $\neg c \vee f$,
$\neg b \vee f$, $\neg d \vee f$.

Indeed, starting with literal a, we get at line 6 $\texttt{bcp}(\Phi \cup \{a\}) = \{a, \neg b, \neg c, \neg d, e\}$. Hence $\mathcal{P}_a = \{\neg a, \neg b, \neg c, \neg d, e\}$. Now, $\texttt{bcp}(\Phi \cup \mathcal{P}_a) = \{\emptyset\}$. As a consequence, one knows at that stage that the equivalence $a \leftrightarrow (\neg b \wedge \neg c \wedge \neg d \wedge e)$ holds in Φ. Then one tries to reduce the number of variables in this AND gate defining a. As explained before, one takes advantage of the implication graph associated with $\texttt{bcp}(\Phi \cup \mathcal{P}_a)$ to do this job. We can observe that the clause $a \vee b \vee c \vee d$ is the unique reason of the conflict. Here, we have $\mathcal{C}_a = \{\neg a, \neg b, \neg c, \neg d\}$. Accordingly, the simpler AND gate $a \leftrightarrow (\neg b \wedge \neg c \wedge \neg d)$ holds in Φ as well. The next step is this to replace a by $\neg b \wedge \neg c \wedge \neg d$ in Φ. This leads to remove the first four clauses of Φ since this replacement turns them into valid clauses. This leads also to turn the clause $\neg a \vee e$ into the clause $b \vee c \vee d \vee e$, and the clause $a \vee f$ into the three clauses $\neg b \vee f$, $\neg c \vee f$, $\neg d \vee f$. Since the size of the resulting formula remains bounded by the size of Φ, the replacement is committed. For the eleven remaining literals, no other AND/OR gate is detected, so that nothing changes since then.

It is easy to show that ANDgateSimpl neither is confluent, nor is projective. To do so, it is sufficient to consider Example 5 again (as already mentioned, literal equivalences can be considered as simple AND/OR gates).

Since each time an AND/OR gate is detected, one variable is eliminated, we get that:

$$\#var(\texttt{ANDgateSimpl}(\Sigma)) \leq \#var(\Sigma).$$

Clearly enough, for each AND/OR gate detected, the replacement of the occurrences of the defined variable by its definition could easily lead to increase the size of the input formula Σ. However, for `ANDgateSimpl`, the replacement of a variable takes place only if it does not lead to increase the size of the formula, so it is ensured that:

$$\#lit(\texttt{ANDgateSimpl}(\Sigma)) \leq \#lit(\Sigma).$$

Like `equivSimpl`, `ANDgateSimpl` may lead to an arbitrarily large tw decrease for some instances but cannot ensure a decrease for each of them. Thus, the same example as the one used for showing that a tw increase may happen can be used again here. Compared to `equivSimpl`, the situation is even worse since the replacement of a variable by the corresponding definition can lead to a treewidth increase, even if the gates are already "syntactically present" in the CNF Σ (i.e., there are clauses in Σ the conjunction of which is equivalent to the gate). The following example shows it. Let $\Sigma_n = \bigwedge_{i=0}^{n-1}(\neg x_{i+1} \vee x_i) \wedge (\neg x_{i+1} \vee y_i) \wedge (\neg x_i \vee \neg y_i \vee x_{i+1})$. Clearly enough, Σ_n is a CNF representation of the conjunction of n AND gates of the form $x_{i+1} \leftrightarrow (x_i \wedge y_i)$ (with i varying from 0 to $n-1$).

Figure 15 gives a representation of the primal graph of Σ_3. We have $tw(\Sigma_3) = 2$.

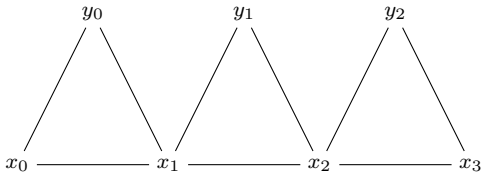

Fig. 15 The primal graph of Σ_3.

Assume that the AND gates are detected in such a way that the gate defining x_i is found before the gate defining x_{i+1} (with i varying from 1 to $n-1$). Each time a gate $x_{i+1} \leftrightarrow (x_0 \wedge \bigwedge_{j=0}^{i} y_j)$ is detected and x_{i+1} is replaced by $x_0 \wedge \bigwedge_{j=0}^{i} y_j$, the treewidth of the resulting CNF formula increases by 1. Thus, at the first step, the gate $x_1 \leftrightarrow (x_0 \wedge y_0)$ is detected and x_1 is replaced by $x_0 \wedge y_0$ in Σ_n. We get $\Sigma_n[x_1 \leftarrow x_0 \wedge y_0] = (\neg x_2 \vee x_0) \wedge (\neg x_2 \vee y_0) \wedge (\neg x_2 \vee y_1) \wedge (\neg x_0 \vee \neg y_0 \vee \neg y_1 \vee x_2)$ $\wedge \bigwedge_{i=2}^{n-1}(\neg x_{i+1} \vee x_i) \wedge (\neg x_{i+1} \vee y_i) \wedge (\neg x_i \vee \neg y_i \vee x_{i+1})$.

Figure 16 gives a representation of the primal graph of $\Sigma_3[x_1 \leftarrow x_0 \wedge y_0])$. We can check that $tw(\Sigma_3[x_1 \leftarrow x_0 \wedge y_0]) = 3$. At the next step the gate $x_2 \leftrightarrow x_0 \wedge y_0 \wedge y_1$ is detected and x_2 is replaced by $x_0 \wedge y_0 \wedge y_1$ in $\Sigma_n[x_1 \leftarrow x_0 \wedge y_0]$. The resulting CNF formula $(\Sigma[x_1 \leftarrow x_0 \wedge y_0])[x_2 \leftarrow x_0 \wedge y_0 \wedge y_1]$ contains the clause $(\neg x_0 \vee \neg y_0 \vee \neg y_1 \vee \neg y_2 \vee x_3)$.

Figure 17 gives a representation of the primal graph of $(\Sigma_3[x_1 \leftarrow x_0 \wedge y_0])[x_2 \leftarrow x_0 \wedge y_0 \wedge y_1]$. We have $tw((\Sigma_3[x_1 \leftarrow x_0 \wedge y_0])[x_2 \leftarrow x_0 \wedge y_0 \wedge y_1]) = 4$. An easy

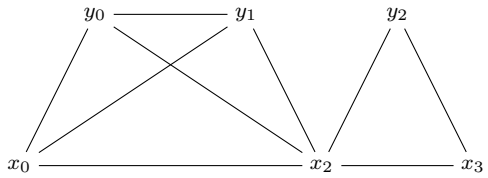

Fig. 16 The primal graph of $\Sigma_3[x_1 \leftarrow x_0 \wedge y_0]$.

structural induction shows that when the AND gates defining x_1, ..., x_{n-1} have been successively detected and the defined variable replaced by its definition, the resulting CNF formula has a treewidth equal to $n + 1$.

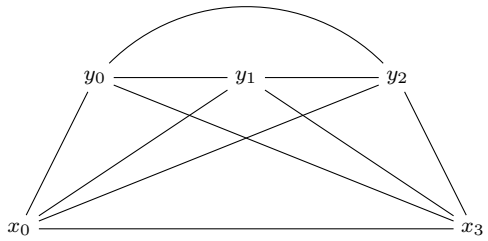

Fig. 17 The primal graph of $(\Sigma_3[x_1 \leftarrow x_0 \wedge y_0])[x_2 \leftarrow x_0 \wedge y_0 \wedge y_1]$.

Indeed, after k successive detections of the gates $x_1 \leftrightarrow x_0 \wedge y_0$, $x_2 \leftrightarrow x_0 \wedge y_0 \wedge y_1$, ..., $x_k \leftrightarrow x_0 \wedge \bigwedge_{j=0}^{k-1} y_j$ followed by the immediate replacement of the defined variable by its definition, the resulting formula

$$(...(\Sigma_n[x_1 \leftarrow x_0 \wedge y_0])[x_2 \leftrightarrow x_0 \wedge y_0 \wedge y_1])...)[x_k \leftrightarrow x_0 \wedge \bigwedge_{j=0}^{k-1} y_j]$$

contains the clause

$$\neg x_0 \vee \bigvee_{j=0}^{k} \neg y_j \vee x_{k+1}$$

as an essential prime implicate. Since every CNF representation of the resulting formula must include that clause, no equivalence-preserving additional preprocessing would prevent from the treewidth increase.

Empirically, we obtained the results reported at Figures 18, 19, and 20.

Figures 18, 19, and 20 clearly show that ANDgateSimpl may lead in practice to very significant reductions of both the $\#var$ value, the $\#lit$ value and the tw value of the instance. The reductions achieved are often much larger than those obtained using the other preprocessing techniques, reflecting the fact that AND/OR gates can be quite numerous in the instances. ANDgateSimpl has an impact on every family

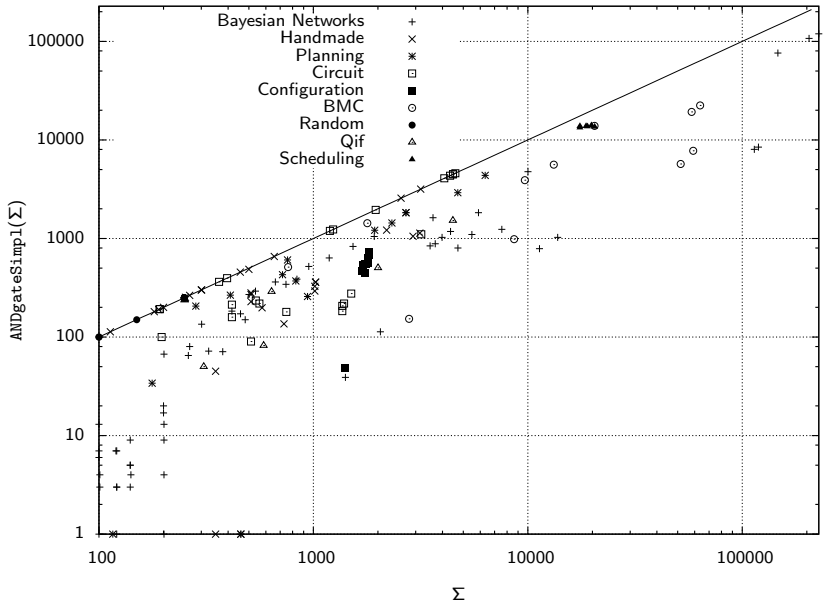

Fig. 18 Comparing $\#var(\Sigma)$ with $\#var(\texttt{ANDgateSimpl}(\Sigma))$.

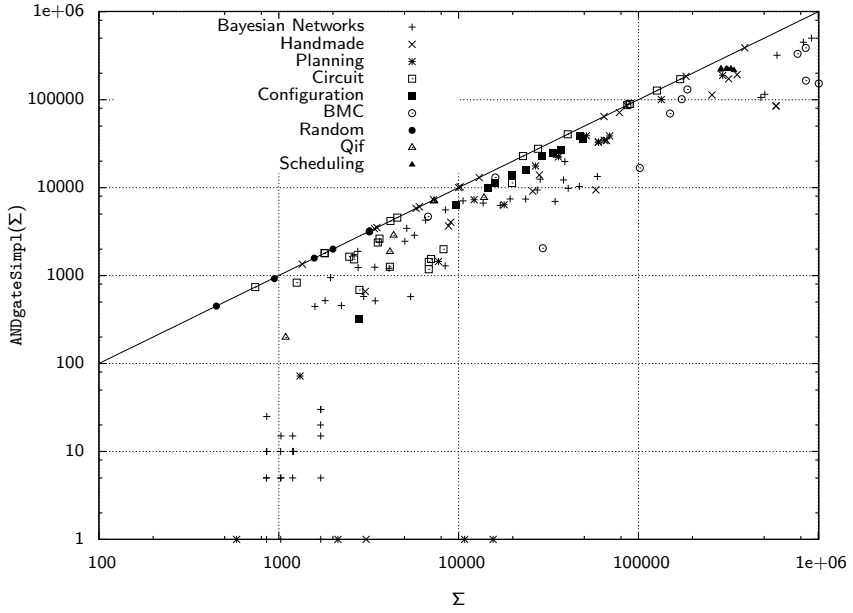

Fig. 19 Comparing $\#lit(\Sigma)$ with $\#lit(\texttt{ANDgateSimpl}(\Sigma))$.

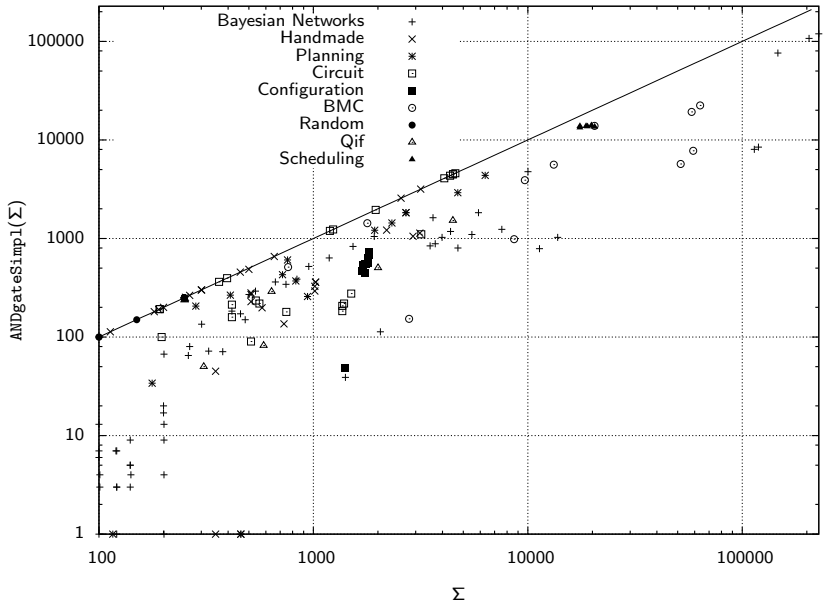

Fig. 20 Comparing $tw(\Sigma)$ with $tw(\texttt{ANDgateSimpl}(\Sigma))$.

of instances considered here, and it appears as quite huge on the Bayesian networks data set. Using the `ANDgateSimpl` preprocessing also improves both the number of instances for which `QuickBB` succeeded in computing a tw value (it terminated with a memory-out for 41 instances over 182) and the number of instances (57) for which `QuickBB` succeeded in computing the exact value of the treewidth. As for `equivSimpl`, no increase of the tw score has resulted in practice from applying `ANDgateSimpl`.

3.2.3 XOR Gates

XOR gate detection and replacement is presented at Algorithm 5. At line 2 some XOR gates $\ell_i \leftrightarrow \chi_i$ are first detected "syntactically" from Σ (i.e., one looks in Σ for the clauses obtained by turning $\ell_i \leftrightarrow \chi_i$ into an equivalent `CNF`; only XOR clauses χ_i of size $\leq maxX$ are targeted; in our experiments $maxX = 5$). Then 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 (once this is done one does not need to replace ℓ_i by its definition in Γ during the replacement step). The last phase is the replacement one (lines 4 and 5): every ℓ_i is replaced by its definition χ_i in Σ, provided that the normalization it involves does not generate "large" clauses (i.e., with size $> maxX$). Due to this condition and the fact that the detection of XOR gates is "syntactic" (i.e., we determine for each clause α of Σ whether it participates to a XOR gate by looking for other clauses of Σ such that,

together with α, form a CNF representation of a XOR gate), the time complexity of XORgateSimpl is in the worst case quadratic in the input size.

Algorithm 6: XORgateSimpl

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

1 $\Phi \leftarrow \Sigma$;
// detection
2 $\Gamma \leftarrow$ {XOR clauses syntactically detected};
// Gaussian elimination
3 $\Gamma \leftarrow$ Gauss($\{\ell_1 \leftrightarrow \chi_1, \ell_2 \leftrightarrow \chi_2, \ldots, \ell_k \leftrightarrow \chi_k\}$)
// replacement
4 **for** $i \leftarrow 1$ *to* k **do**
5 **if** $\nexists \alpha \in \Phi[\ell_i \leftarrow \chi_i] \setminus \Phi$ *s.t.* $|\alpha| > maxX$ **then** $\Phi \leftarrow \Phi[\ell_i \leftarrow \chi_i]$;
6 **return** Φ

Example 7 Let Σ be the CNF formula consisting of the following clauses:

$b \vee d$,
$\neg b \vee \neg d$,
$\neg a \vee \neg b \vee c$,
$a \vee \neg b \vee \neg c$,
$\neg a \vee b \vee \neg c$,
$a \vee b \vee c$,
$b \vee e$,
$a \vee f$.

At line 2, the two XOR gates $b \oplus d$ and $a \oplus b \oplus c$ are detected successively and put in a set Γ. Then at line 3, Gauss elimination is achieved in this set, leading to the two XOR gates $b \oplus d$ and $a \oplus \neg d \oplus c$. At line 5 the replacement step is done in the input formula: the first six clauses which participate to the XOR gates are made valid through this substitution, 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 $\neg c \vee \neg d \vee f$. Finally, XORgateSimpl$(\Sigma)$ consists of the following clauses:

$\neg d \vee e$,
$c \vee d \vee f$,
$\neg c \vee \neg d \vee f$.

For the same reasons as those pointed out for equivSimpl and ANDgateSimpl, XORgateSimpl neither is confluent nor is projective. To prove it, it is enough to consider Example 5 again (literal equivalences can also be considered as XOR gates, and in Example 5 all the gates about variables a, b, c, d can be detected "syntactically").

Since each time a XOR gate is detected, one variable is eliminated, we get that:

$$\#var(\texttt{XORgateSimpl}(\Sigma)) \leq \#var(\Sigma).$$

Now, unlike equivSimpl and ANDgateSimpl, applying XORgateSimpl to Σ may lead to increase the number of literals in it. However, only a "reasonable" increase is allowed due to the presence of the $maxX$ condition in Algorithm 6. More

precisely, the size of the output formula Φ is upper bounded by $|\Sigma|.\ 2^{maxX-1}.maxX$, which is acceptable in practice when $maxX$ is small enough.

Finally, like `equivSimpl` and `ANDgateSimpl`, `XORgateSimpl` may offer but does not ensure a tw decrease in the general case. A similar family of `CNF` formulae as the one considered for `ANDgateSimpl` can be used to show a similar result when XOR gates are considered (basically, Σ_n is then a `CNF` representation of the conjunction of n XOR gates of the form $x_{i+1} \leftrightarrow (x_i \leftrightarrow y_i)$, with i varying from 1 to $n-1$).

Empirically, we obtained the results reported at Figures 21, 22 and 23.

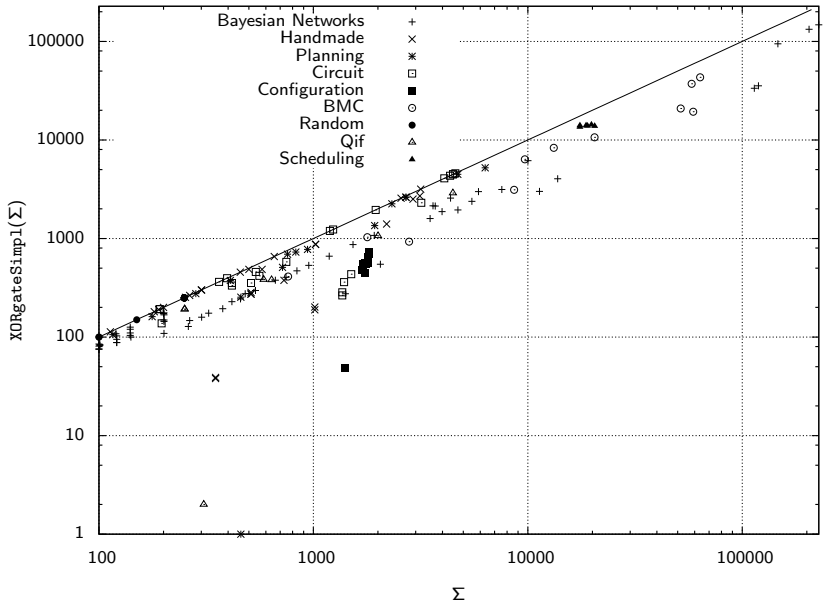

Fig. 21 Comparing $\#var(\Sigma)$ with $\#var(\texttt{XORgateSimpl}(\Sigma))$.

One can observe on Figures 21, 22 and 23 that `XORgateSimpl` may lead in practice to reductions of both the $\#var$ value, the $\#lit$ value and the tw value of the instance. However, the number of instances for which huge reductions (i.e., one order of magnitude and above) are obtained is typically lower than the number of instances for which huge reductions are obtained when other elementary preprocessings (but `equivSimpl`) are considered. Note also that using the `XORgateSimpl` preprocessing improves both the number of instances for which `QuickBB` succeeded in computing a tw value (it terminated with a memory-out for 45 instances over 182) and the number of instances (20) for which `QuickBB` succeeded in computing the exact value of the treewidth. Finally, applying `XORgateSimpl` never led to increase the tw scores of the instances considered in the experiments.

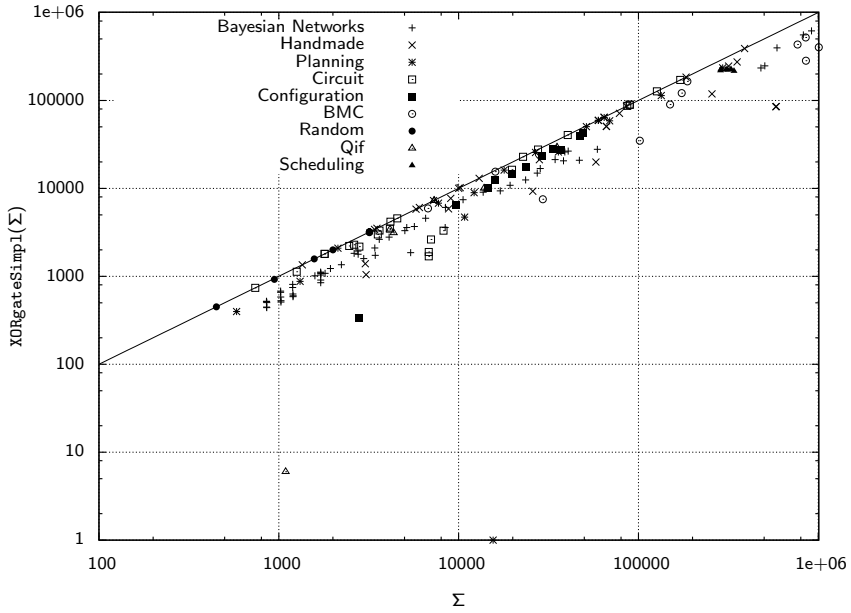

Fig. 22 Comparing $\#lit(\Sigma)$ with $\#lit(\texttt{XORgateSimpl}(\Sigma))$.

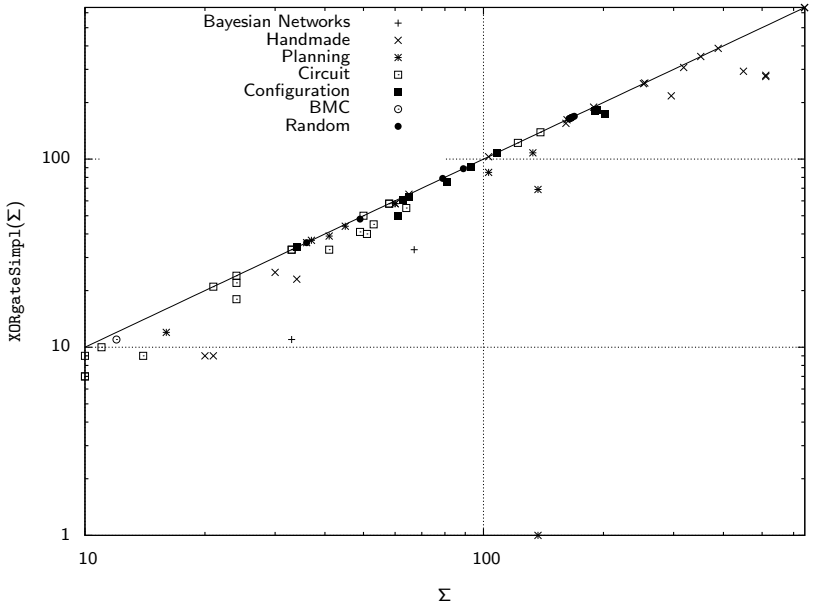

Fig. 23 Comparing $tw(\Sigma)$ with $tw(\texttt{XORgateSimpl}(\Sigma))$.

3.3 The pmc Preprocessor, the eq Combination, and the $\#eq$ Combination

Our preprocessor pmc (cf. Algorithm 7) is based on the elementary preprocessing techniques presented before. Each elementary technique is invoked or not, depending on the value of a Boolean parameter: $optV$ (vivification), $optB$ (backbone identification), $optO$ (occurrence reduction), $optG$ (gate detection and replacement). gatesSimpl(Φ) is a short for XORgateSimpl(ANDgateSimpl(equivSimpl(Φ))).

pmc is an iterative algorithm. Indeed, it can prove useful to apply more than once some elementary techniques since each application may change the resulting CNF formula. This is not the case for backbone identification, and this explains why it is performed at start, only. Indeed, each of the remaining techniques generates a CNF formula which is a logical consequence of its input. As a consequence, if a literal belongs to the backbone of a CNF formula which results from the composition of such elementary preprocessings, then it belongs as well to the backbone of the CNF formula considered initially. Any further call to backboneSimpl would just be a waste of time.

Once the literals of the backbone have been detected and propagated using BCP, we use occurrenceSimpl in order to possibly reduce the size of the input clauses, thus enhancing the power of BCP which is used next for discovering literal equivalences and gates. vivificationSimpl could be used as well, but since it may lead to remove some clauses, it does not empower BCP as occurrenceSimpl does. Thus we use vivificationSimpl at the end of each iteration to propagate in the whole CNF the effect of the equivalences and gates replacement and so to say to "clean" the instance.

Within pmc the elementary preprocessings (but backboneSimpl) can be performed several times. Iteration stops when a fixed point is reached (i.e., the output of the preprocessing is equal to its input) or when a preset (maximal) number $numTries$ of iterations is reached. In our experiments $numTries$ was set to 10.

Algorithm 7: pmc

```
   input  : a CNF formula Σ
   output : a CNF formula Φ such that ||Φ|| = ||Σ||
 1 Φ←Σ;
 2 if optB then Φ←backboneSimpl(Φ);
 3 i←0;
 4 while i < numTries do
 5     i←i + 1;
 6     if optO then Φ← occurrenceSimpl(Φ);
 7     if optG then Φ←gatesSimpl(Φ);
 8     if optV then Φ←vivificationSimpl(Φ);
 9     if fixpoint then break;
10 return Φ
```

In our work, we have considered two combinations of the elementary preprocessings described in the previous section:

- 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.

Example 8 Let Σ be the `CNF` formula Σ consisting of the following clauses:

$b_1 \vee b_2 \vee b_3$, $b_1 \vee b_2 \vee \neg b_3$, $b_1 \vee \neg b_2 \vee b_3$, $b_1 \vee \neg b_2 \vee \neg b_3$, $\neg b_1 \vee v_1 \vee o_1$, $v_5 \vee o_2$, $\neg o_1 \vee \neg o_2 \vee v_1$, $v_6 \vee o_3$, $v_{10} \vee o_6$, $v_6 \vee v_{10} \vee \neg o_6$, $v_1 \vee v_2 \vee v_3 \vee \neg v_4$, $v_1 \vee v_5 \vee o_5$, $v_2 \vee \neg v_5 \vee v_{11}$, $v_6 \vee v_7 \vee v_8 \vee \neg v_9$, $v_6 \vee v_{10} \vee o_6$, $v_7 \vee \neg v_{10} \vee \neg v_8$, $\neg e_1 \vee e_2$, $\neg e_2 \vee e_3$, $\neg e_3 \vee e_1$, $\neg b_1 \vee e_2 \vee e_3 \vee \neg e_4$, $\neg e_2 \vee e_3 \vee \neg e_5$, $a_1 \vee \neg e_2 \vee \neg e_4$, $e_4 \vee \neg e_2 \vee \neg a_1$, $\neg e_4 \vee a_2$, $x_1 \vee x_2 \vee x_3$, $\neg x_1 \vee \neg x_2 \vee x_3$, $\neg x_1 \vee x_2 \vee \neg x_3$, $x_1 \vee \neg x_2 \vee \neg x_3$, $\neg i_1 \vee \neg e_2 \vee a_2$, $\neg i_1 \vee e_4$, $i_1 \vee \neg a_2$.

This example aims at illustrating the effect on Σ of the successive elementary preprocessings that have been considered when the $\#eq$ combination is used. The variable names refer to the preprocessing which is illustrated. b variables are for "backbone detection", o variables for "occurrence simplification", v variables for "vivification", e variables for "equivalence detection and replacement", a variables for "AND gate detection and replacement", x variables for "XOR gate detection and replacement", and finally i variables are useful for illustrating the impact of putting together and performing iteratively some preprocessings (as it is the case in the two combinations eq and $\#eq$ we have defined).

The backbone of Σ is equal to $B = \{b_1\}$. `backboneSimpl`(Σ) consists of the following clauses:

b_1, $v_1 \vee o_1$, $v_5 \vee o_2$, $\neg o_1 \vee \neg o_2 \vee v_1$, $v_6 \vee o_3$, $v_{10} \vee o_6$, $v_6 \vee v_{10} \vee \neg o_6$, $v_1 \vee v_2 \vee v_3 \vee \neg v_4$, $v_1 \vee v_5 \vee o_5$, $v_2 \vee \neg v_5 \vee v_{11}$, $v_6 \vee v_7 \vee v_8 \vee \neg v_9$, $v_6 \vee v_{10} \vee o_6$, $v_7 \vee \neg v_{10} \vee \neg v_8$, $\neg e_1 \vee e_2$, $\neg e_2 \vee e_3$, $\neg e_3 \vee e_1$, $e_2 \vee e_3 \vee \neg e_4$, $\neg e_2 \vee e_3 \vee \neg e_5$, $a_1 \vee \neg e_2 \vee \neg e_4$, $e_4 \vee \neg e_2 \vee \neg a_1$, $\neg e_4 \vee a_2$, $x_1 \vee x_2 \vee x_3$, $\neg x_1 \vee \neg x_2 \vee x_3$, $\neg x_1 \vee x_2 \vee \neg x_3$, $x_1 \vee \neg x_2 \vee \neg x_3$, $\neg i_1 \vee \neg e_2 \vee a_2$, $\neg i_1 \vee e_4$, $i_1 \vee \neg a_2$.

`occurrenceSimpl`(`backboneSimpl`(Σ)) consists of the following clauses:

b_1, $v_1 \vee o_1$, $v_5 \vee o_2$, $\neg o_1 \vee \neg o_2 \vee v_1$, $v_6 \vee o_3$, $v_{10} \vee o_6$, $v_6 \vee v_{10} \vee \neg o_6$, $v_1 \vee v_2 \vee v_3 \vee \neg v_4$, $v_1 \vee v_5 \vee o_5$, $v_2 \vee \neg v_5 \vee v_{11}$, $v_6 \vee v_7 \vee v_8 \vee v_9$, $v_6 \vee v_{10}$, $v_7 \vee \neg v_{10} \vee \neg v_8$, $\neg e_1 \vee e_2$, $\neg e_2 \vee e_3$, $\neg e_3 \vee e_1$,

$e_2 \vee e_3 \vee \neg e_4$, $e_4 \vee \neg e_2 \vee \neg a_1$, $\neg x_1 \vee \neg x_2 \vee x_3$, $\neg i_1 \vee \neg e_2 \vee a_2$,
$\neg e_2 \vee e_3 \vee \neg e_5$, $\neg e_4 \vee a_2$, $\neg x_1 \vee x_2 \vee \neg x_3$, $\neg i_1 \vee e_4$,
$a_1 \vee \neg e_2 \vee \neg e_4$, $x_1 \vee x_2 \vee x_3$, $x_1 \vee \neg x_2 \vee \neg x_3$, $i_1 \vee \neg a_2$.

The effect of `occurrenceSimpl` has been to reduce $v_1 \vee v_5 \vee o_5$ to $v_1 \vee v_5$, and to reduce $v_6 \vee v_{10} \vee o_6$ to $v_6 \vee v_{10}$.

`vivificationSimpl`(`occurrenceSimpl` (`backboneSimpl`(Σ))) consists of the following clauses:

b_1, $v_2 \vee \neg v_5 \vee v_{11}$, $e_2 \vee e_3 \vee \neg e_4$, $\neg x_1 \vee x_2 \vee \neg x_3$,
$v_1 \vee o_1$, $v_6 \vee v_7 \vee v_9$, $\neg e_2 \vee e_3 \vee \neg e_5$, $x_1 \vee \neg x_2 \vee \neg x_3$,
$v_5 \vee o_2$, $v_6 \vee v_{10}$, $a_1 \vee \neg e_2 \vee \neg e_4$, $\neg i_1 \vee \neg e_2 \vee a_2$,
$\neg o_1 \vee \neg o_2 \vee v_1$, $v_7 \vee \neg v_{10} \vee \neg v_8$, $e_4 \vee \neg e_2 \vee \neg a_1$, $\neg i_1 \vee e_4$,
$v_6 \vee o_3$, $\neg e_1 \vee e_2$, $\neg e_4 \vee a_2$, $i_1 \vee \neg a_2$.
$v_{10} \vee o_6$, $\neg e_2 \vee e_3$, $x_1 \vee x_2 \vee x_3$,
$v_1 \vee v_2 \vee v_3 \vee \neg v_4$, $\neg e_3 \vee e_1$, $\neg x_1 \vee \neg x_2 \vee x_3$,

The effect of `vivificationSimpl` has been to cancel the clauses $v_6 \vee v_{10} \vee \neg o_6$ and $v_1 \vee v_5$, and to reduce $v_6 \vee v_7 \vee v_8 \vee v_9$ to $v_6 \vee v_7 \vee v_9$.

`equivSimpl`(`vivificationSimpl` (`occurrenceSimpl`(`backboneSimpl`(Σ)))) consists of the following clauses:

b_1, $v_2 \vee \neg v_5 \vee v_{11}$, $\neg e_4 \vee a_2$, $i_1 \vee \neg a_2$.
$v_1 \vee o_1$, $v_6 \vee v_7 \vee v_9$, $x_1 \vee x_2 \vee x_3$,
$v_5 \vee o_2$, $v_6 \vee v_{10}$, $\neg x_1 \vee \neg x_2 \vee x_3$,
$\neg o_1 \vee \neg o_2 \vee v_1$, $v_7 \vee \neg v_{10} \vee \neg v_8$, $\neg x_1 \vee x_2 \vee \neg x_3$,
$v_6 \vee o_3$, $e_2 \vee \neg e_4$, $x_1 \vee \neg x_2 \vee \neg x_3$,
$v_{10} \vee o_6$, $a_1 \vee \neg e_2 \vee \neg e_4$, $\neg i_1 \vee \neg e_2 \vee a_2$,
$v_1 \vee v_2 \vee v_3 \vee \neg v_4$, $e_4 \vee \neg e_2 \vee \neg a_1$, $\neg i_1 \vee e_4$,

The literals e_1, e_2 and e_3 are found equivalent using `bcp`. Thus, every occurrence of e_1 or e_3 is replaced by e_2. The effect of `equivSimpl` has been to reduce $e_2 \vee e_3 \vee \neg e_4$ into $e_2 \vee \neg e_4$ and to cancel the three clauses $\neg e_1 \vee e_2$, $\neg e_2 \vee e_3$, and $\neg e_3 \vee e_1$ (the replacement step makes each of them valid), as well as the clause $\neg e_2 \vee e_3 \vee \neg e_5$ (again, replacing e_3 by e_2 in it generates a valid clause).

`ANDgateSimpl`(`equivSimpl`(`vivificationSimpl` (`occurrenceSimpl`(`backboneSimpl`(Σ))))) consists of the following clauses:

b_1, $v_{10} \vee o_6$, $v_7 \vee \neg v_{10} \vee \neg v_8$, $x_1 \vee \neg x_2 \vee \neg x_3$,
$v_1 \vee o_1$, $v_1 \vee v_2 \vee v_3 \vee \neg v_4$, $\neg e_2 \vee \neg a_1 \vee a_2$, $\neg i_1 \vee \neg e_2 \vee a_2$,
$v_5 \vee o_2$, $v_2 \vee \neg v_5 \vee v_{11}$, $x_1 \vee x_2 \vee x_3$, $\neg i_1 \vee e_2$,
$\neg o_1 \vee \neg o_2 \vee v_1$, $v_6 \vee v_7 \vee v_9$, $\neg x_1 \vee \neg x_2 \vee x_3$, $\neg i_1 \vee a_1$,
$v_6 \vee o_3$, $v_6 \vee v_{10}$, $\neg x_1 \vee x_2 \vee \neg x_3$, $i_1 \vee \neg a_2$.

Using bcp, the AND gate $e_4 \leftrightarrow (e_2 \wedge a_1)$ is first detected. Observe that the power of unit propagation is used to this purpose; especially the clause $a_1 \vee \neg e_4$ which is obtained by turning the gate into CNF does not appear in the input; only the clause $a_1 \vee \neg e_4 \vee \neg e_2$ is available; the fact that e_2 can be obtained from the input and e_4 using bcp is exploited again to derive a_1 from e_4 and e_2. Then the effect of ANDgateSimpl has been to replace e_4 by its definition $e_2 \wedge a_1$ everywhere in the input. In the three clauses $e_2 \vee \neg e_4$, $a_1 \vee \neg e_2 \vee \neg e_4$, and $e_4 \vee \neg e_2 \vee \neg a_1$ which have been used to find the definition, the replacement leads to valid clauses, which are thus cancelled. Finally, the replacement of e_4 by its definition in $\neg e_4 \vee a_2$ leads to the clause $\neg e_2 \vee \neg a_1 \vee a_2$ and in $\neg i_1 \vee e_4$ leads to the clauses $\neg i_1 \vee e_2$ and $\neg i_1 \vee a_1$.

XORgateSimpl(ANDgateSimpl(equivSimpl (vivificationSimpl(occurrenceSimpl(backboneSimpl(Σ)))))) consists of the following clauses:

b_1,	$v_6 \vee o_3$,	$v_6 \vee v_7 \vee v_9$,	$\neg i_1 \vee \neg e_2 \vee a_2$,
$v_1 \vee o_1$,	$v_{10} \vee o_6$,	$v_6 \vee v_{10}$,	$\neg i_1 \vee e_2$,
$v_5 \vee o_2$,	$v_1 \vee v_2 \vee v_3 \vee \neg v_4$,	$v_7 \vee \neg v_{10} \vee \neg v_8$,	$\neg i_1 \vee a_1$,
$\neg o_1 \vee \neg o_2 \vee v_1$,	$v_2 \vee \neg v_5 \vee v_{11}$,	$\neg e_2 \vee \neg a_1 \vee a_2$,	$i_1 \vee \neg a_2$.

The XOR gate $x_1 \leftrightarrow (\neg x_2 \oplus x_3)$ is first detected. Then the effect of XORgateSimpl has been to cancel the four clauses $x_1 \vee x_2 \vee x_3$, $\neg x_1 \vee \neg x_2 \vee x_3$, $\neg x_1 \vee x_2 \vee \neg x_3$, and $x_1 \vee \neg x_2 \vee \neg x_3$, which are used to show that x_1 is equivalent to $\neg x_2 \oplus x_3$.

Finally, pmc(Σ) consists of the following clauses:

b_1,	$\neg o_1 \vee \neg o_2 \vee v_1$,	$v_1 \vee v_2 \vee v_3 \vee \neg v_4$,	$v_6 \vee v_{10}$,
$v_1 \vee o_1$,	$v_6 \vee o_3$,	$v_2 \vee \neg v_5 \vee v_{11}$,	$v_7 \vee \neg v_{10} \vee \neg v_8$.
$v_5 \vee o_2$,	$v_{10} \vee o_6$,	$v_6 \vee v_7 \vee v_9$,	

The fixed point is reached after 3 iterations. During the second iteration, the equivalence $i_1 \leftrightarrow a_2$ is detected by equivSimpl. This leads first to cancel the clauses $\neg i_1 \vee \neg e_2 \vee a_2$ and $i_1 \vee \neg a_2$. Furthermore, the replacement of i_1 by its definition in $\neg i_1 \vee e_2$ leads to generate the clause $\neg a_2 \vee e_2$; the replacement of i_1 by its definition in $\neg i_1 \vee a_1$ leads to generate the clause $\neg a_2 \vee a_1$. Finally, during the third iteration, the AND gate $a_2 \leftrightarrow (e_2 \wedge a_1)$ is detected. The corresponding replacement leads to cancel the clauses $\neg e_2 \vee \neg a_1 \vee a_2$, $\neg a_2 \vee e_2$ and $\neg a_2 \vee a_1$, which have been used to find the definition.

3.3.1 The eq Combination

Clearly enough, the *eq* preprocessing ensures that:

$$\#var(eq(\Sigma)) \leq \#var(\Sigma),$$

$$\#lit(eq(\Sigma)) \leq \#lit(\Sigma),$$

$$tw(eq(\Sigma)) \leq tw(\Sigma)$$

since every elementary preprocessing in it satisfies such inequalities.

Empirically, we obtained the results reported at Figures 24, 25, and 26.

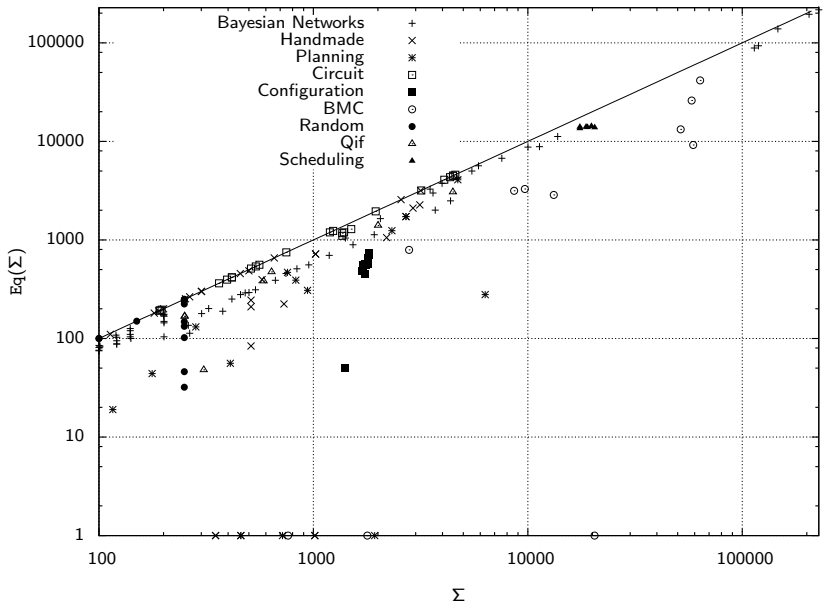

Fig. 24 Comparing $\#var(\Sigma)$ with $\#var(eq(\Sigma))$.

Figures 24, 25, and 26 show significant improvements of both the $\#var$ value, the $\#lit$ value and the tw value of the instance. Especially, some synergetic effects obtained by combining elementary preprocessings can be observed; for instance, for the configuration data set, taking advantage of eq leads to enhanced reductions of the $\#lit$ value and the tw value, compared with those obtained using `backboneSimpl` alone. The eq preprocessing also improves both the number of instances for which `QuickBB` succeeded in computing a tw value (it terminated with a memory-out for 43 instances over 182) and the number of instances (48) for which `QuickBB` succeeded in computing the exact value of the treewidth.

3.3.2 The $\#eq$ Combination

Contrariwise to the eq preprocessing , the $\#eq$ preprocessing ensures only that:

$$\#var(\#eq(\Sigma)) \leq \#var(\Sigma).$$

For the $\#lit$ measure and for the tw measure, an increase is possible.

Empirically, we obtained the results reported at Figures 27, 28, and 29.

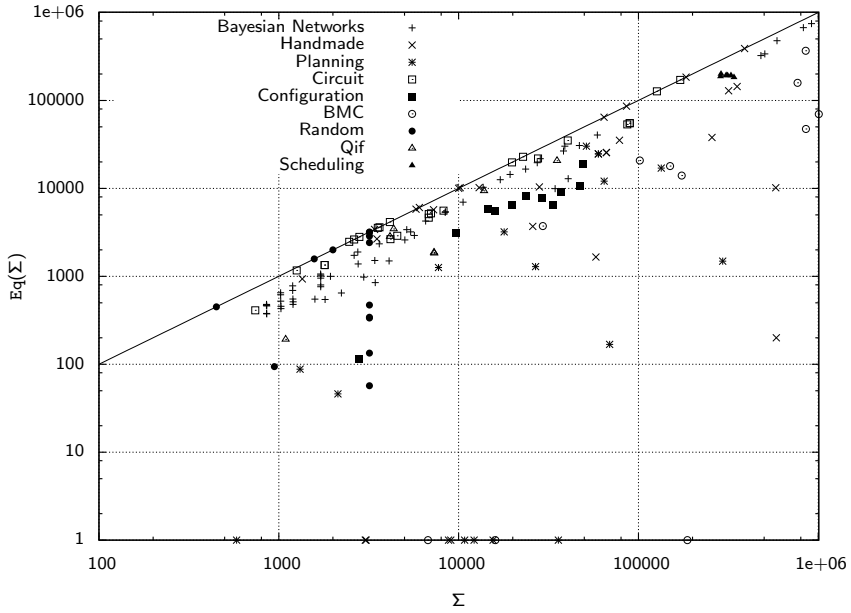

Fig. 25 Comparing $\#lit(\Sigma)$ with $\#lit(eq(\Sigma))$.

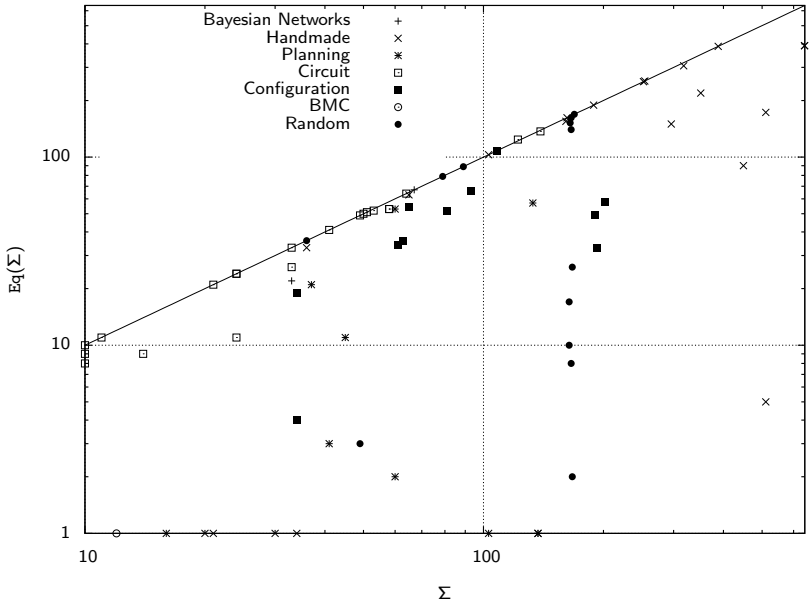

Fig. 26 Comparing $tw(\Sigma)$ with $tw(eq(\Sigma))$.

Those figures show that $\#eq$ may lead in practice to huge reductions of both the $\#var$ value, the $\#lit$ value and the tw value of the instance. Again, some synergetic effects obtained by combining elementary preprocessings can be observed; compared with `ANDgateSimpl` (which is the most efficient elementary preprocessing included in the $\#eq$ combination), the reduction of the $\#var$ value of the instance is more important (whatever the data set under consideration). For the configuration data set and the circuit data set, the further decreases of the $\#lit$ value and of the tw value obtained by exploiting $\#eq$ are also salient. Unsurprisingly $\#eq$ appears as the most efficient preprocessing among those we considered (in the sense that it leads to the most significant reductions of the $\#var$ value, the $\#lit$ value and the tw value). As the other preprocessings, using $\#eq$ also improves both the number of instances for which `QuickBB` succeeded in computing a tw value (it terminated with a memory-out for 29 instances over 182) and the number of instances (79) for which `QuickBB` succeeded in computing the exact value of the treewidth.

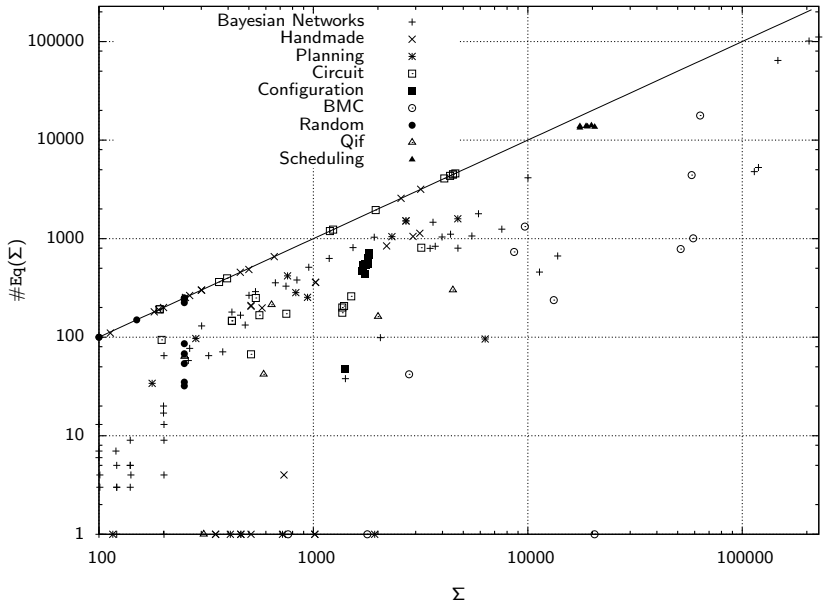

Fig. 27 Comparing $\#var(\Sigma)$ with $\#var(\#eq(\Sigma))$.

4 On the Impact of Preprocessings on Model Counting Techniques

In a second step, we evaluated the impact of each elementary preprocessing by coupling it with a model counter. Two families of model counters have been considered downstream.

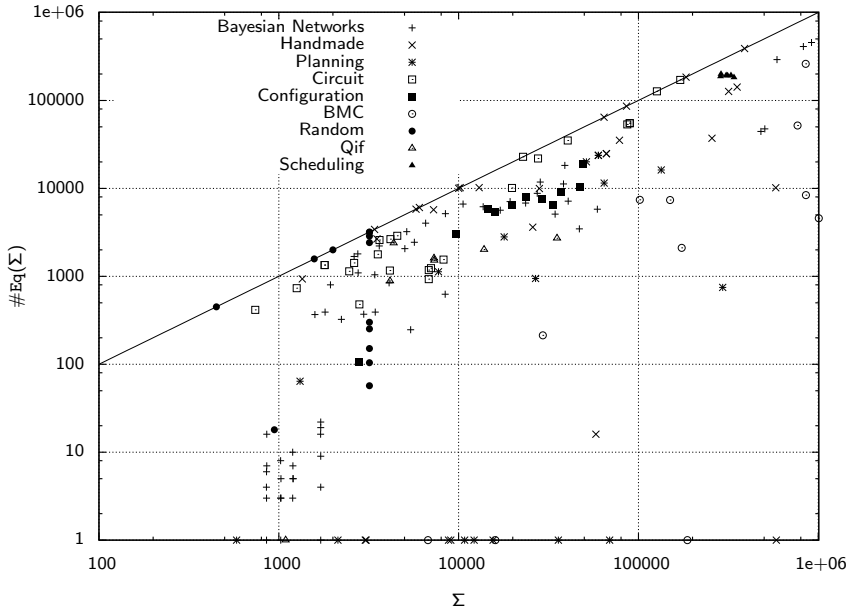

Fig. 28 Comparing $\#lit(\Sigma)$ with $\#lit(\#eq(\Sigma))$.

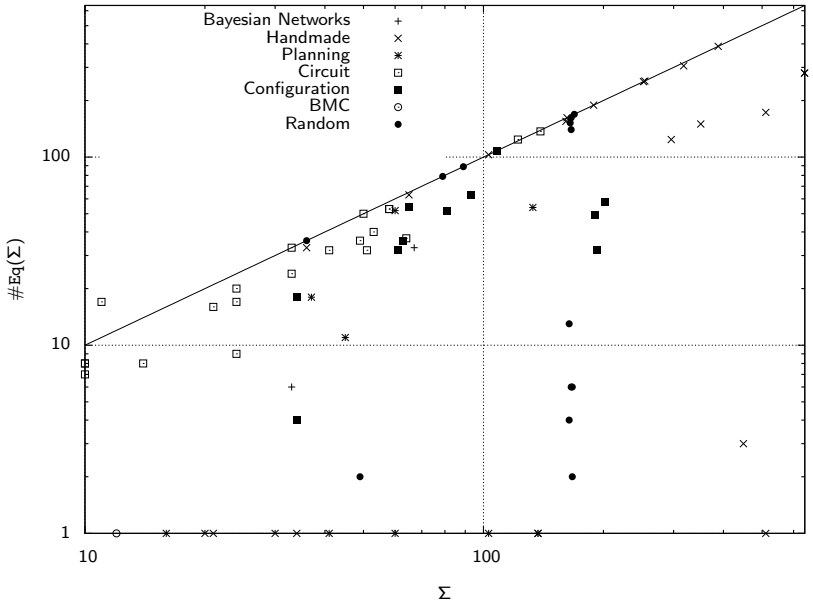

Fig. 29 Comparing $tw(\Sigma)$ with $tw(\#eq(\Sigma))$.

We first considered "direct" model counters, i.e., without any compilation. We took advantage of the exact model counters Cachet [50] and sharpSAT [55], as well as of the approximate model counter SampleCount [25].

The second family gathers compilation-based approaches to model counting. We considered the compilers C2D [14,15], Dsharp [43], SDD [16], and cnf2obdd [56, 31]. C2D and Dsharp target the Decision-DNNF language; SDD and cnf2obdd target respectively the languages SDD and OBDD (two subsets of the d−DNNF language).

We considered the 182 instances already used in the previous section. All the model counters and compilers have been run with their options by default, and a time limit of 1h per instance as been considered for each of them, but SampleCount. For SampleCount, the cutoff has been fixed at 200000. Only equivalence-preserving techniques (including the eq combination) have been used when the compilation-based approaches have been considered.

For each model counting technique used, for each preprocessing technique p under consideration, we computed the numbers of benchmarks "solved"[4] when p is used, and when it is not. The results are presented under the form of cactus plots. Such plots make precise the number of instances "solved" in a given amount of time per instance.

4.1 Exact and Approximate Model Counters

4.1.1 Cachet

Cachet (www.cs.rochester.edu/~kautz/Cachet/index.htm) [50] is an exact model counter, based on the ZChaff SAT solver [42,59], and integrating component caching with clause learning. The branching heuristic it exploits is Variable State Aware Decaying Sum (VSADS) [51].

Figure 30 illustrates the impact of each preprocessing considered in the previous section on Cachet. We can observe that the best performers are backboneSimpl, eq, ANDgateSimpl, and $\#eq$. Clearly enough, $\#eq$ leads to the larger improvements, and appears as significantly better than backboneSimpl, eq, and ANDgateSimpl. The other preprocessings do not have any noticeable impact (their level of performance is close to the one obtained when no preprocessing is used).

4.1.2 sharpSAT

sharpSAT (sites.google.com/site/marcthurley/sharpsat) [55] also is an exact model counter. Compared to Cachet, sharpSAT takes advantage of a new approach of coding components, which reduces the cache size significantly, and a new cache management scheme. Furthermore, within sharpSAT, (implicit) bcp is used at every decision point in order to find some failed literals, i.e., literals which are falsified in every model of the formula corresponding to the decision point. This is done independently of the variable selected by the branching heuristic (which

[4] "Solved" means that the number of models has been found for all methods but SampleCount, and that an approximation has been computed when SampleCount is considered.

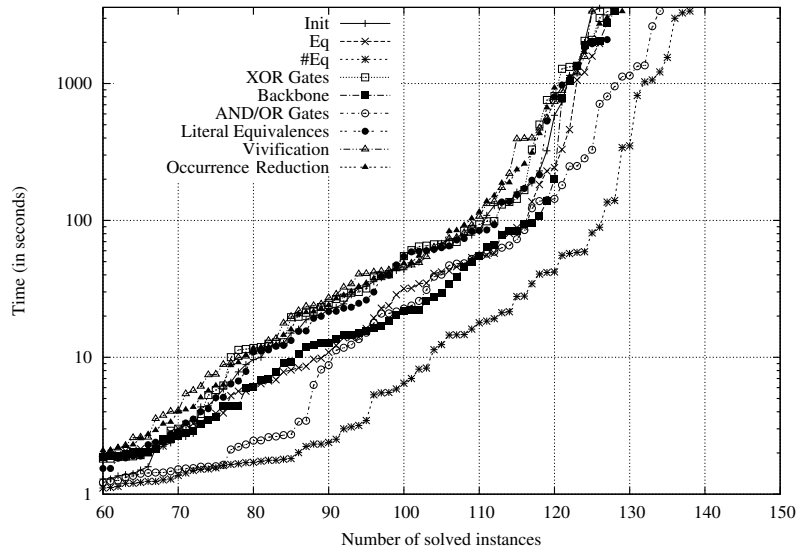

Fig. 30 Comparing the impact of preprocessings on `Cachet`.

is VSADS, as in `Cachet`). The literals which belong to clauses that became binary in the most recent call to `bcp` are selected as candidates for failed literals. A conflict clause is learned for each failed literal found.

Figure 31 presents similar results as those presented for `Cachet`, but based on the model counter `sharpSAT`. Again, $\#eq$ leads to the larger improvements. While `backboneSimpl`, eq, `ANDgateSimpl` exhibit quite the same level of performance as when `Cachet` is used downstream, it turns out that `ANDgateSimpl` is significantly better than eq when the model counter under consideration is `sharpSAT`. Interestingly, every preprocessing considered here has an impact (it leads to solve more quickly more instances than those solved when no preprocessing takes place).

4.1.3 `SampleCount`

`SampleCount` (`www.cs.cornell.edu/~sabhar/#software`) [25] is an approximate model counter. It is a randomized algorithm with a high probability of success. Indeed, `SampleCount` provides provably (probabilistic) guaranteed lower bounds on the model counts of the input formula using solution sampling. More precisely, `SampleCount` first uses sampling to select a set of variables of the formula to fix. Once a sufficient number of variables have been set, the remaining formula can be counted using an exact model counter (`Cachet`). From the exact residual model count and the number of fixed variables, a lower bound on the total number of models is obtained.

Figure 32 presents the numbers of instances for which `SampleCount` succeeded in computing an approximation of the number of models, for a cutoff equal to 200000, depending on the chosen preprocessing.

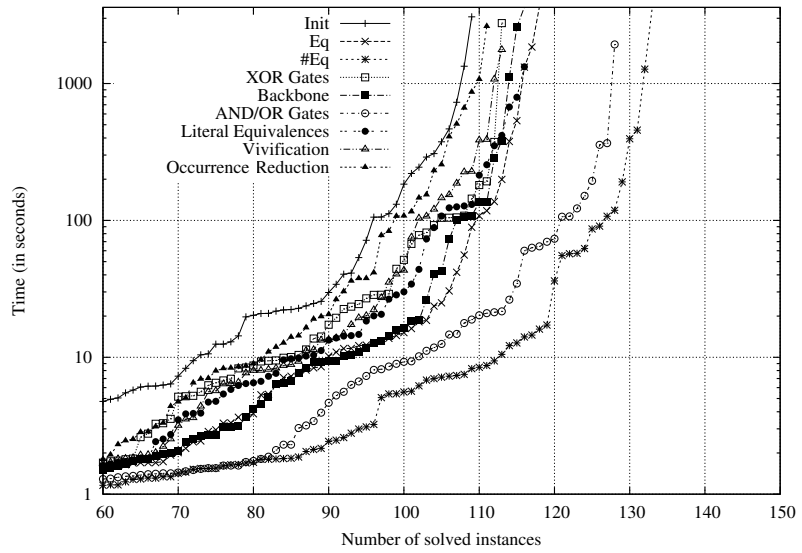

Fig. 31 Comparing the impact of preprocessings on `sharpSAT`.

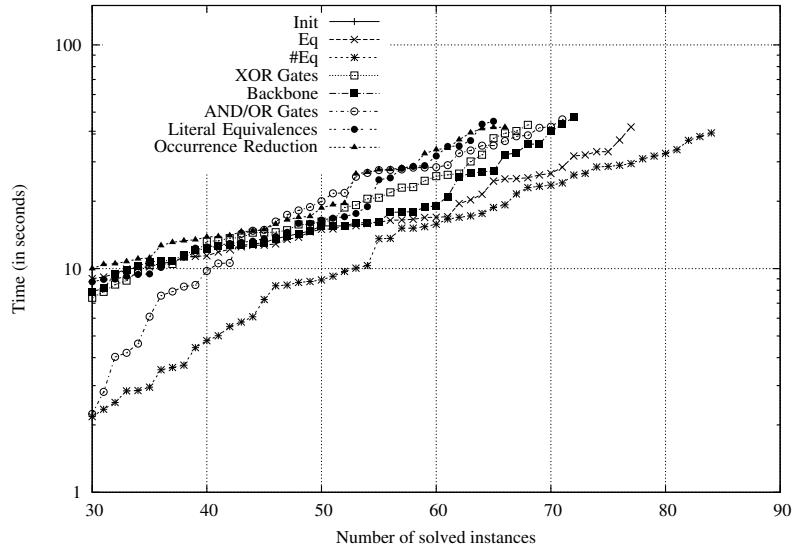

Fig. 32 Comparing the numbers of instances solved by `SampleCount` equipped or not with some preprocessings.

$\#eq$ appears as the best preprocessing technique among those tested as to the number of instances "solved". However, the very significance of this measure is doubtful, since it does not tell anything about the quality of the approximation. In-

deed, 0 is a lower bound of the number of models of any instance, and it can be computed in constant time! For this reason, we performed some additional experiments in order to determine whether applying some preprocessing technique leads or not to improve the approximation of the number of models achieved by `SampleCount`. The results are reported in Table 1. Each cell indicates the number of instances for which `SampleCount` equipped with the preprocessing technique associated with the row led to a better estimate (i.e., a larger value since `SampleCount` reports a lower bound of the number of models of the input instance) than `SampleCount` equipped with the preprocessing technique associated with the column. "Init" means that no preprocessing has been used. Interestingly, the "Init" column of the top table shows that each preprocessing has a positive influence on the quality of the lower bounds computed using `SampleCount`; furthermore, the $\#eq$ rows show that $\#eq$ is the approach leading to the best improvements among the tested preprocessings.

	Init	`backboneSimpl`	`occurrenceSimpl`	`vivificationSimpl`
Init	0	-57	-37	-47
`backboneSimpl`	57	0	20	22
`occurrenceSimpl`	37	-20	0	1
`vivificationSimpl`	47	-22	-1	0
`equivSimpl`	39	-18	6	-5
`ANDgateSimpl`	52	-16	10	11
`XORgateSimpl`	37	-20	2	3
eq	56	1	15	19
$\#eq$	76	3	36	39

	`equivSimpl`	`ANDgateSimpl`	`XORgateSimpl`	eq	$\#eq$
Init	-39	-52	-37	-56	-76
`backboneSimpl`	18	16	20	-1	-3
`occurrenceSimpl`	-6	-10	-2	-15	-36
`vivificationSimpl`	5	-11	-3	-19	-39
`equivSimpl`	0	-14	2	-15	-33
`ANDgateSimpl`	14	0	13	-7	-26
`XORgateSimpl`	-2	-13	0	-15	-33
eq	15	7	15	0	-19
$\#eq$	33	26	33	19	0

Table 1 Quality of the estimates of the number of models reported by `SampleCount` depending on the preprocessing which has been applied first.

4.2 Compilation-Based Model Counting

4.2.1 `C2D`

`C2D` (`reasoning.cs.ucla.edu/c2d/`) [14, 15] is a top-down compiler targeting the Decision-`DNNF` language. The generation of the decision nodes in the resulting Decision-`DNNF` representation is guided by a decomposition tree (dtree) of the input `CNF` instance Σ, which is computed first.

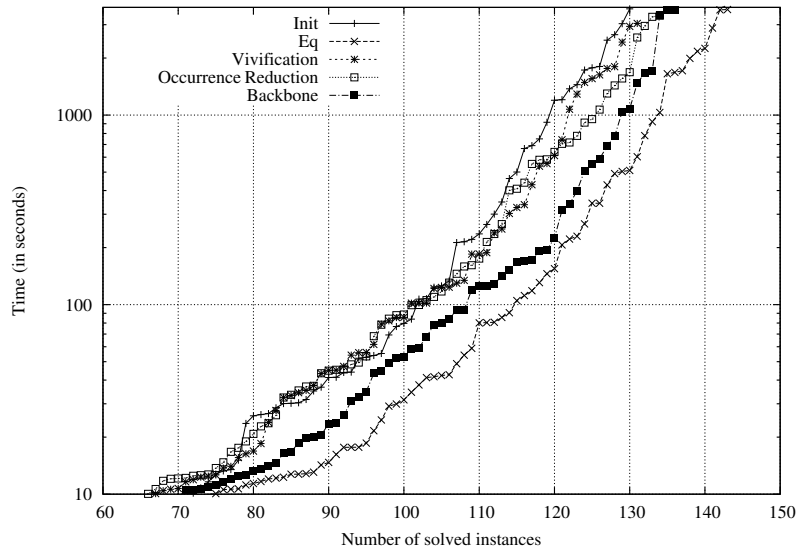

Fig. 33 Comparing the impact of preprocessings on C2D.

Figure 33 gives the numbers of instances which have been successfully compiled by C2D within the time limit, depending on the preprocessing used. *eq* appears as the best performer, offering larger gains than backboneSimpl, which led itself to a larger number of solved instances than the other preprocessing techniques. For them, the impact is unclear (they exhibit quite the same level of performance as when no preprocessing is performed).

4.2.2 Dsharp

Dsharp (www.haz.ca/research/dsharp/) [43] also is a top-down compiler targeting the Decision-DNNF language. The Decision-DNNF representations which are generated by Dsharp are generated by following the traces of the sharpSAT model counter, thus following the approach presented in [31]. Unlike C2D, the generation of the decision nodes is not guided by a decomposition tree, but a dynamic decomposition approach is used for performing the disjoint component analysis. Dsharp also incorporates some preprocessing technique, namely a restricted form of backbone detection, based on bcp.

Figure 34 presents similar results as above, but based on the Dsharp compiler. This time, both backboneSimpl and *eq* appear as the best performers, offering larger gains than vivificationSimpl and occurrenceSimpl. Interestingly, every preprocessing technique considered here has an impact.

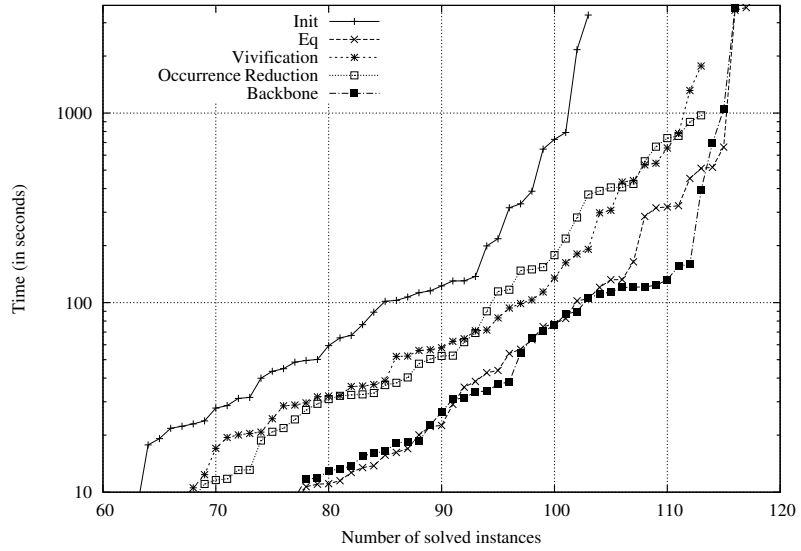

Fig. 34 Comparing the impact of preprocessings on `Dsharp`.

4.2.3 `SDD`

`SDD` (`reasoning.cs.ucla.edu/sdd/` [16] is a bottom-up compiler targeting the `SDD` language. A variable tree (vtree) associated with the `CNF` instance is searched in a dynamic way, and an `SDD` representation of Σ for this tree is generated. This representation is canonical (each computed `SDD` representation is compressed and trimmed).

Figure 35 presents similar results as above for the `SDD` compiler. The same observations as those made for `Dsharp` can also be done. The improvements offered by `backboneSimpl` and eq over `vivificationSimpl` and `occurrenceSimpl`, and by `vivificationSimpl` and `occurrenceSimpl` over the case when no preprocessing is used are nevertheless more prominent than for `Dsharp`.

4.2.4 `cnf2obdd`

`cnf2obdd` (`www.sd.is.uec.ac.jp/toda/code/cnf2obdd.html`) [56] is a top-down `OBDD` compiler, following the approach presented in [31].

Figure 36 presents similar results as above for the `cnf2obdd` compiler. The same observations as for `SDD` can be made.

5 A Large-Scale Evaluation of eq and $\#eq$

Finally, we performed a large-scale evaluation of the benefits offered by the eq and $\#eq$ combinations, for each of the model counting technique considered in the pre-

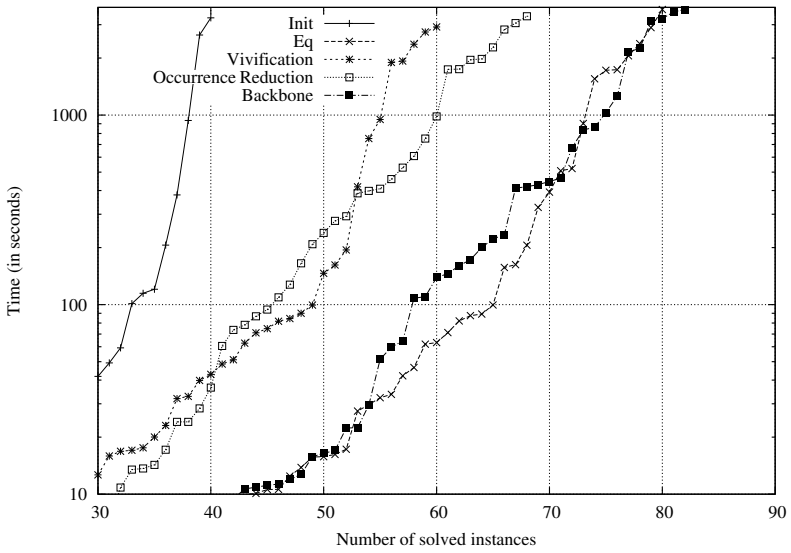

Fig. 35 Comparing the impact of preprocessings on SDD.

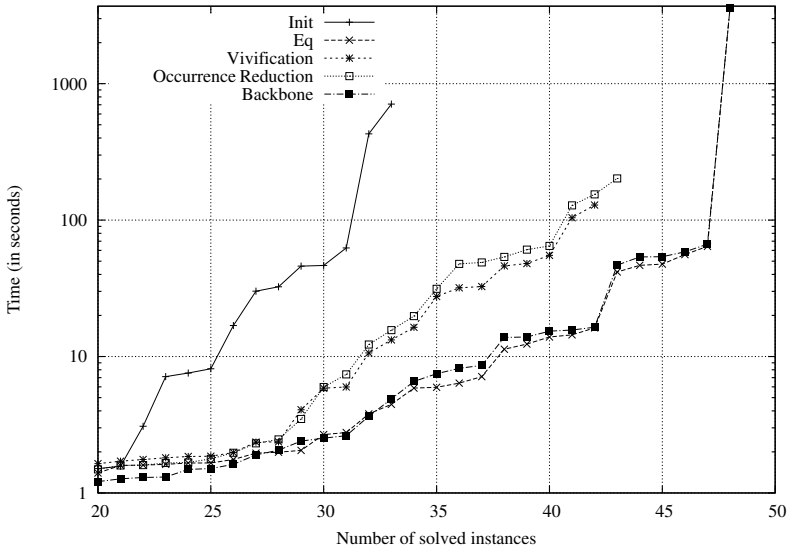

Fig. 36 Comparing the impact of preprocessings on cnf2obdd.

vious section. We made quite intensive experiments on a number of CNF instances Σ from different domains, available again in the SAT LIBrary. 1449 instances Σ (from the same 9 families as those considered in the previous sections) have been used. They are gathered as follows:

- BMC (18)
- Circuit (68)
- Qif (7)
- Planning (34)
- Random (105)
- Scheduling (6)
- Handmade (58)
- Configuration (35)
- Bayesian networks (1118)

No specific optimization of the preprocessing achieved depending on the family of the instance under consideration has been considered.

We have first performed some experiments for computing the simplification times required by the two combinations of preprocessings eq and $\#eq$, on those instances. It turns out that `pmc` is not time-expensive: the instances can typically be preprocessed using a few milliseconds (even when they contain many variables and/or many clauses).

Then, the aim was to count the number of models of each instance Σ using `pmc` for the two combinations of preprocessings eq and $\#eq$, and to determine whether the combination(s) under consideration prove(s) or not useful for this task.

For the direct model counters except `SampleCount`, we compared for each instance the time needed to solve it (i.e., to compute the number of models) when no preprocessing was used, with the time needed to solve it when eq (resp. $\#eq$) has been applied first (of course, the preprocessing time is part of the global solving time), and we also compared eq with $\#eq$. The time limit and memory limit considered in the experiments are the same ones as those considered in the previous experiments (1h and 7.6 GiB).

As to `SampleCount`, we compared the quality of the aproximations of the number of models obtained (with a cutoff equal to 200000) when no preprocessing, eq or $\#eq$ were first used.

For the compilation-based model counters, we performed a similar comparison as in the case of exact and direct model counters, yet focusing on the compilation times and the sizes of the resulting compiled forms.

The results are mainly presented using scatter plots (i.e., for all model counters but `SampleCount`). Each dot corresponds to a `CNF` instance Σ. For `Cachet` and `sharpSAT`, the x-axis of the figure represents the computation time needed to count its number of models when no preprocessing has been performed (or when eq has been applied first), while the y-axis represents the time needed to count its number of models when eq (or $\#eq$) has been applied first. For the compilation-based model counters, the x-axis of the figure represents the compilation time (or the size of the compiled form) when no preprocessing has been performed, while the y-axis represents the compilation time (or the size of the compiled form) when eq has been applied first.

5.1 On Simplification Times

For each of the two combinations eq and $\#eq$, Figures 37 and 38 indicate (respectively) how many instances over 1449 are preprocessed within a given time limit (in seconds). The two cactus plots are very similar (we report them in two distinct pictures for readability reasons). This similarity shows that the gate detection and replacement step is very efficient. Going into more detail, 78% (resp. 77 %) of the instances are preprocessed within 1s when eq (resp. $\#eq$) is considered; 94% (resp. 94 %) of the instances are preprocessed within 10s when eq (resp. $\#eq$) is considered. 99% (resp. 99 %) of the instances are preprocessed within 100s when eq (resp. $\#eq$) is considered. Ten (resp. eleven) instances required more than 100s for being preprocessed when eq (resp. $\#eq$) is considered. For four of them the preprocessing phase did not terminate before the time-out of 3600s. For the great majority of instances Σ for which `pmc` requires more than 1s, almost all the computation time is spent in `backboneSimpl`; those instances are typically hard for SAT, and unsurprisingly, none of the downstream model counters has proved able to solve them (whatever they have been preprocessed or not). The reported simplification times appear as very small most of the time for the two combinations under consideration. Hence, for sure, they are very small as well for all the elementary preprocessings used in them (but `backboneSimpl` in some cases).

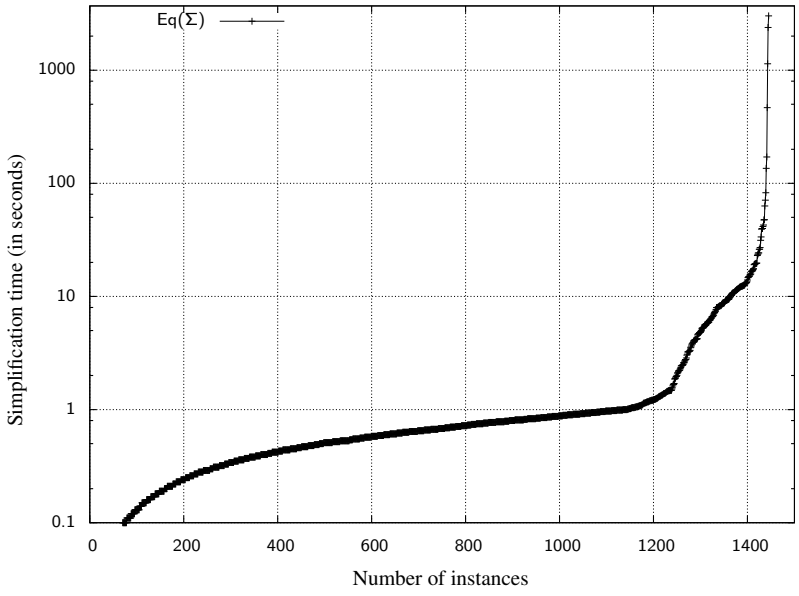

Fig. 37 Number of instances (over 1449) processed by `pmc` equipped with the *eq* combination in a given time bound.

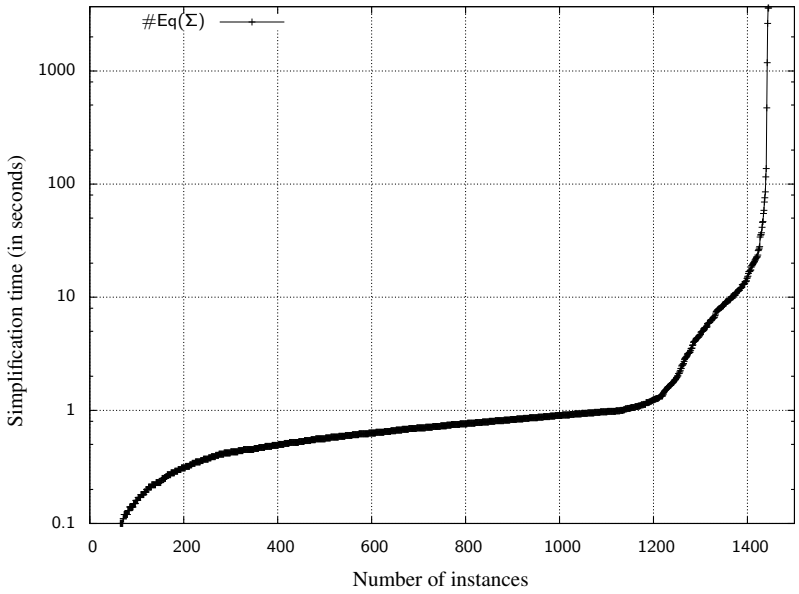

Fig. 38 Number of instances (over 1449) processed by `pmc` equipped with the $\#eq$ combination in a given time bound.

5.2 Exact and Approximate Model Counters

5.2.1 `Cachet`

Figure 39 presents the obtained results for `Cachet` when the eq preprocessing is used. The solving times decrease often, especially this is frequent for the instances from the random data set. Improvements are also obtained for many instances from the Bayesian networks data set. We can also observe that it is not rare that applying the eq preprocessing leads to increase the computation times spent by `Cachet` for counting the number of models of the instance.

Figure 40 presents similar results for `Cachet` when the $\#eq$ preprocessing is used. This time, huge time savings are obtained for most of the families of instances. Compared to the eq preprocessing, the number of instances for which using $\#eq$ leads to decrease significantly the performances of the model counter is small. The benefits offered by $\#eq$ over eq are also salient on Figure 41. Globally, `Cachet` equipped with $\#eq$ (resp. eq, no preprocessing) has been able to solve 1221 instances (resp. 1111, 1083) over 1449 within the time and memory limits.

5.2.2 `sharpSAT`

Figures 42, 43, and 44 reports similar results as those obtained for `Cachet`, but considering the `sharpSAT` model counter instead. The observations we can do are

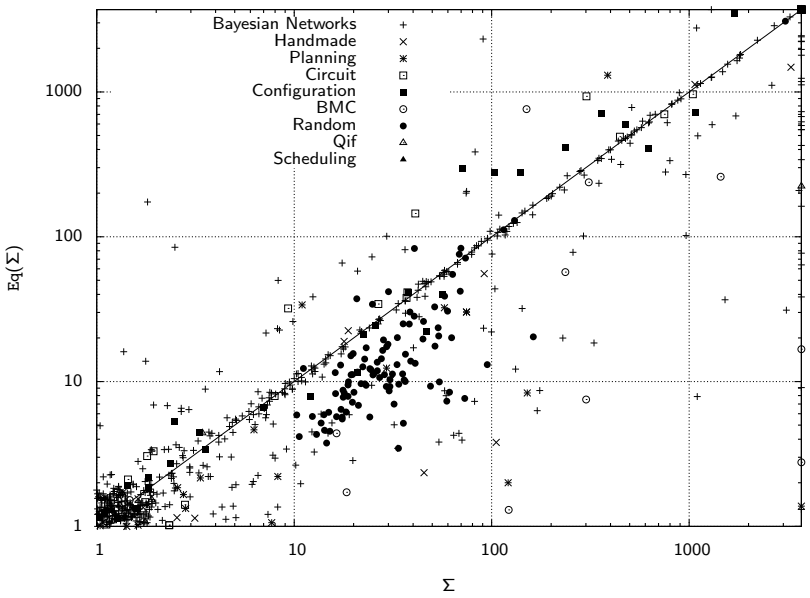

Fig. 39 Comparison of the computation times needed to count the number of models of an instance using `Cachet`, when no preprocessing is used vs. when the *eq* combination has been applied first.

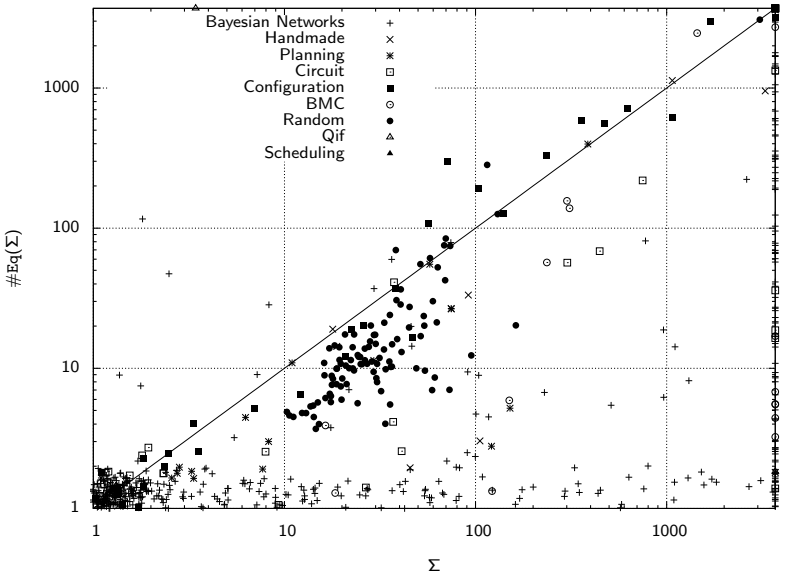

Fig. 40 Comparison of the computation times needed to count the number of models of an instance using `Cachet`, when no preprocessing is used vs. when the $\#eq$ combination has been applied first.

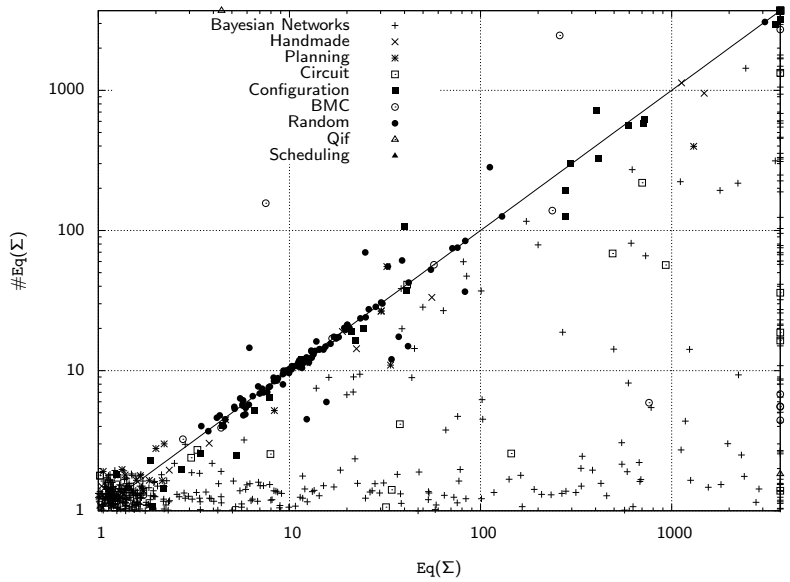

Fig. 41 Comparison of the computation times needed to count the number of models of an instance using `Cachet`, when the eq combination has been applied first vs. when the $\#eq$ combination has been applied first.

globally the same ones as those made for `Cachet`, except that the benefits offered by the two preprocessings eq and $\#eq$ are less important than in the case of `Cachet`. The instances for which using eq or $\#eq$ leads to degrade the computation times are also more numerous than for `Cachet`. Nevertheless, `sharpSAT` equipped with $\#eq$ (resp. eq, no preprocessing) has been able to solve 1226 instances (resp. 1114, 1114) over 1449 within the time and memory limits.

5.2.3 `SampleCount`

Table 2 compares the estimates of the number of models reported by `SampleCount` when no preprocessing, the eq combination or the $\#eq$ combination has been applied first. As in Table 1, each cell of the table indicates the number of instances (over 1449) for which `SampleCount` equipped with the preprocessing technique associated with the row led to a better estimate (i.e., a larger value since `SampleCount` reports a lower bound of the number of models of the input instance) than `SampleCount` equipped with the preprocessing technique associated with the column. "Init" means that no preprocessing has been used.

Clearly enough, both eq and $\#eq$ have a positive influence on the quality of the lower bounds computed using `SampleCount`. $\#eq$ leads to better bounds than eq in many cases.

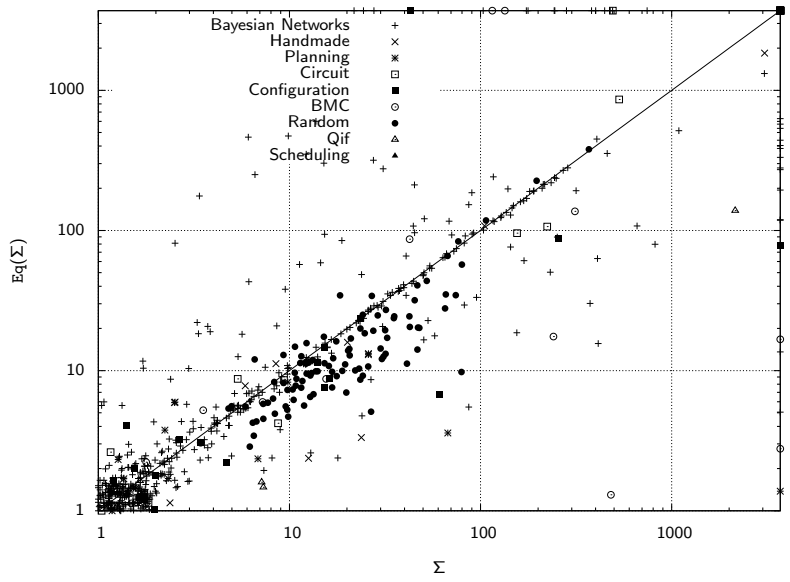

Fig. 42 Comparison of the computation times needed to count the number of models of an instance using `sharpSAT`, when no preprocessing is used vs. when the *eq* combination has been applied first.

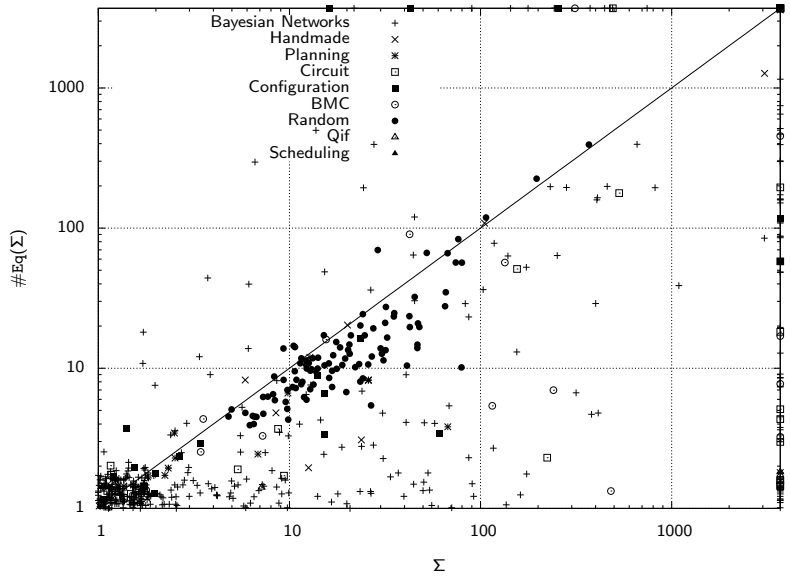

Fig. 43 Comparison of the computation times needed to count the number of models of an instance using `sharpSAT`, when no preprocessing is used vs. when the $\#eq$ combination has been applied first.

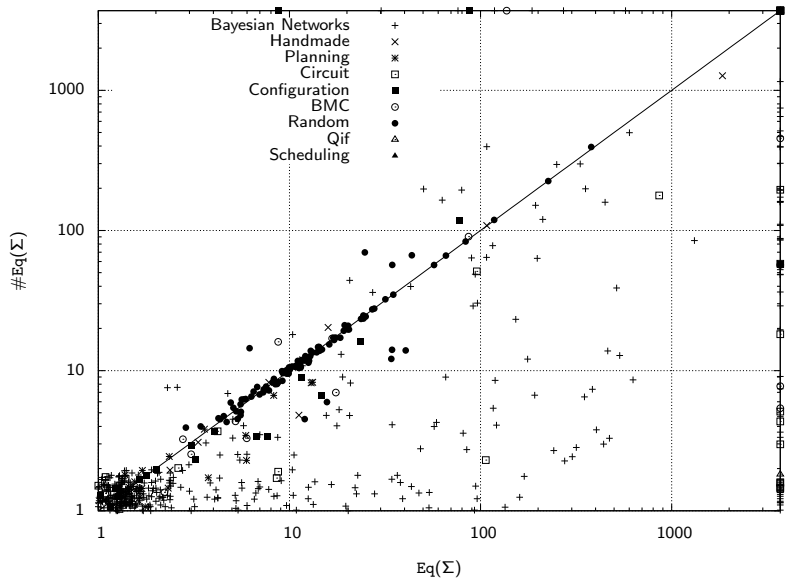

Fig. 44 Comparison of the computation times needed to count the number of models of an instance using `sharpSAT`, when the *eq* combination has been applied first vs. when the $\#eq$ combination has been applied first.

	Init	*eq*	$\#eq$
Init	0	-144	-300
eq	144	0	-155
$\#eq$	300	155	0

Table 2 Quality of the estimates of the number of models reported by `SampleCount` when no preprocessing, the *eq* combination or the $\#eq$ combination has been applied first.

5.3 Compilation-Based Model Counting

5.3.1 `C2D`

Figure 45 presents the compilation times spent by `C2D` when no preprocessing is used and when the *eq* combination has been applied first. The impact of the *eq* preprocessing is salient and concerns all the families of instances.

Figure 46 presents the sizes of the compiled forms obtained using `C2D`, when no preprocessing is used and when the *eq* combination has been applied first. The results cohere with those obtained for the compilation times. The impact of the *eq* preprocessing on the sizes of the resulting Decision-`DNNF` representations appears as very significant. The instances for which the *eq* preprocessing has a negative influence are quite rare, and the corresponding increases in the sizes of the compiled forms remain limited. Globally, `C2D` equipped with *eq* (resp. with no preprocessing) has been able to solve 1329 instances (resp. 1265) over 1449 within the time and memory

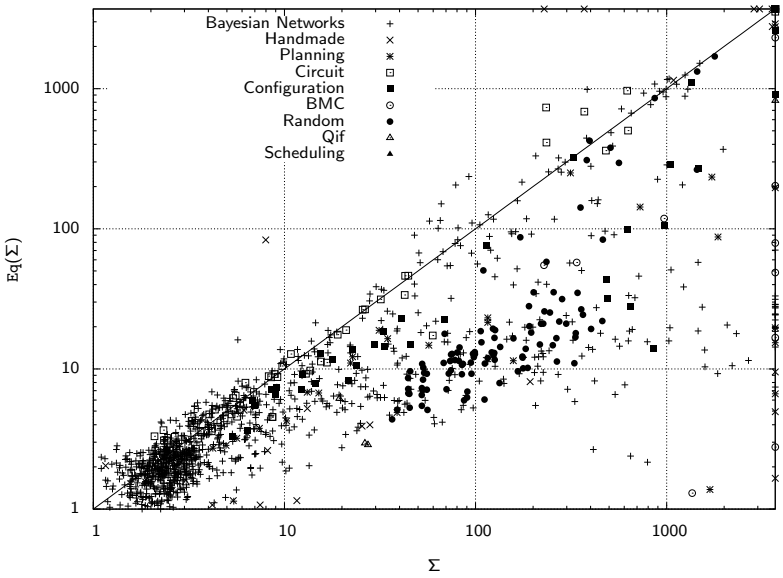

Fig. 45 Comparisons of the compilation times of C2D, when no preprocessing is used vs. when the *eq* combination has been applied first.

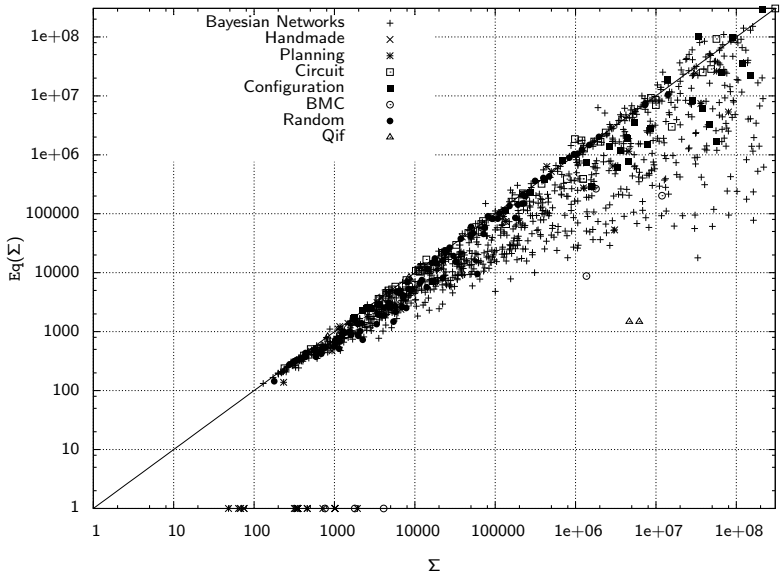

Fig. 46 Comparisons of the sizes of the compiled forms obtained using C2D, when no preprocessing is used vs. when the *eq* combination has been applied first.

limits. It proved empirically as the best approach on the set of benchmarks we used (which can be explained by the huge number of instances from the Bayesian networks family considered in the experiments).

5.3.2 Dsharp

Figures 47 and 48 present similar results when Dsharp is used as the downstream compiler. This time, the improvements are more mitigated. While the *eq* preprocessing has a positive influence on both the compilation times and the sizes of the compiled forms for instances from the random family, it turns out to exhibit a negative influence on those measures for instances from the Bayesian networks data set. For many instances, it can be observed that the *eq* preprocessing has no influence (or a very limited one) on the two measures. This highly contrasts with the observations which can be made when the compiler C2D is used. Dsharp equipped with *eq* (resp. with no preprocessing) has been able to solve 968 instances (resp. 973) over 1449 within the time and memory limits. Thus, the influence of the *eq* preprocessing on Dsharp is globally negative as to the number of instances "solved".

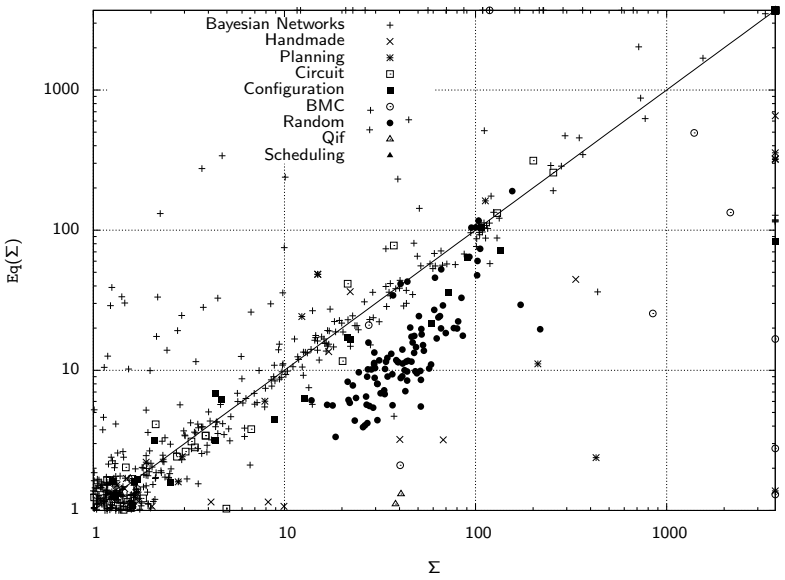

Fig. 47 Comparisons of the compilation times of Dsharp, when no preprocessing is used vs. when the *eq* combination has been applied first.

5.3.3 SDD

Figures 49 and 50 give the results for SDD. The influence of the *eq* preprocessing on the compilation times can be huge, both positively and negatively, especially for instances from the Bayesian networks data set. The influence of the *eq* preprocessing on the sizes of the compiled forms can be positive or negative as well, but it appears as quite limited in most cases, despite the fact that the vtree which is computed can be deeply impacted by the preprocessing. Note nevertheless that *eq* has a clear positive

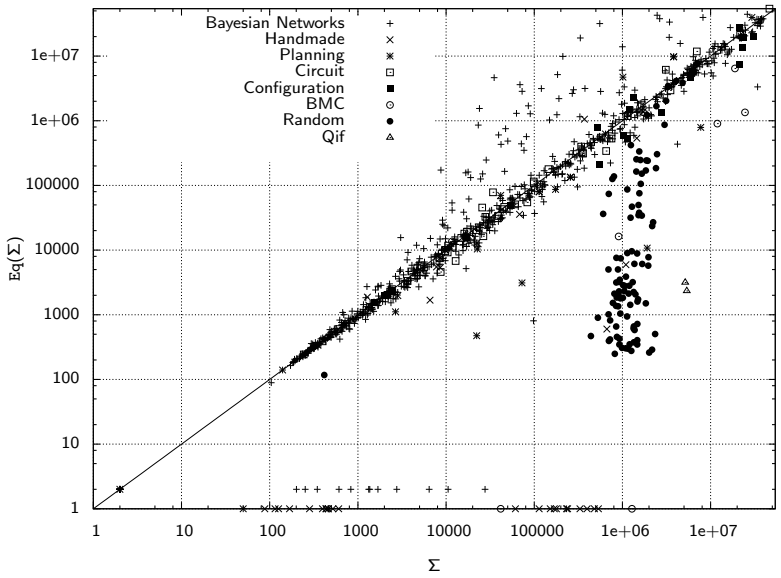

Fig. 48 Comparisons of the sizes of the compiled forms obtained using `Dsharp`, when no preprocessing is used vs. when the *eq* combination has been applied first.

influence on the number of instances solved: `SDD` equipped with *eq* (resp. with no preprocessing) has been able to solve 815 instances (resp. 730) over 1449 within the time and memory limits.

5.3.4 `cnf2obdd`

Finally, Figures 51 and 52 give the results obtained for `cnf2obdd`. The influence of the *eq* preprocessing on the compilation times appears as positive or negative depending on the instances, but is limited for most cases, except for instances from the random family, for which it is typically positive and huge. As to the sizes of the compiled forms, the *eq* preprocessing does not show any significant impact, except for some instances of the random family, for which a large decrease can be observed. Because $\texttt{OBDD}_<$ representations are canonical ones for each variable ordering $<$, no equivalence-preserving preprocessing technique can have an impact of the size of the resulting representation. However, the variable ordering used by `cnf2obdd` is generated using the MINCE heuristic [1], and the modification of the `CNF` instance Σ which results from applying any preprocessing p may easily lead MINCE to generate a variable ordering associated with $p(\Sigma)$ different from the one associated with Σ. This explains why the sizes of the compiled forms produced by *eq*+`cnf2obdd` may differ from those produced by `cnf2obdd`. Globally, `cnf2obdd` equipped with *eq* (resp. with no preprocessing) has been able to solve 304 instances (resp. 261) over 1449 within the time and memory limits.

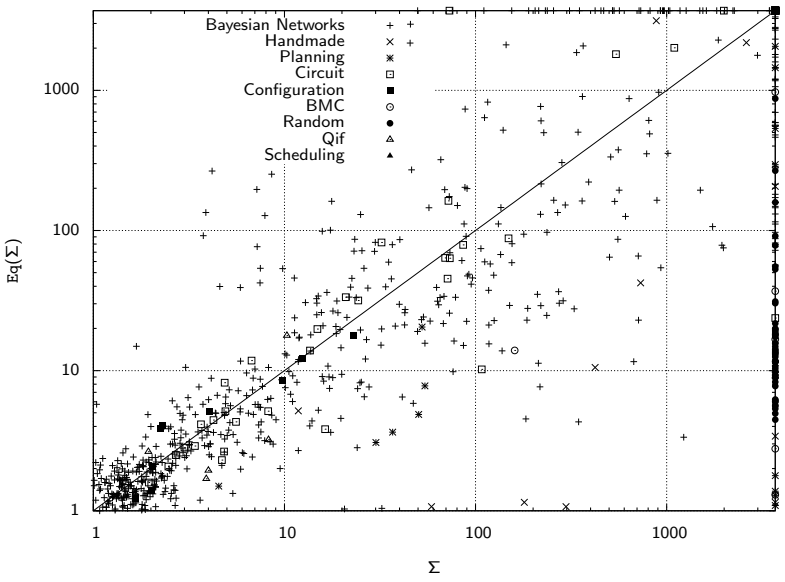

Fig. 49 Comparisons of the compilation times of SDD, when no preprocessing is used vs. when the *eq* combination has been applied first.

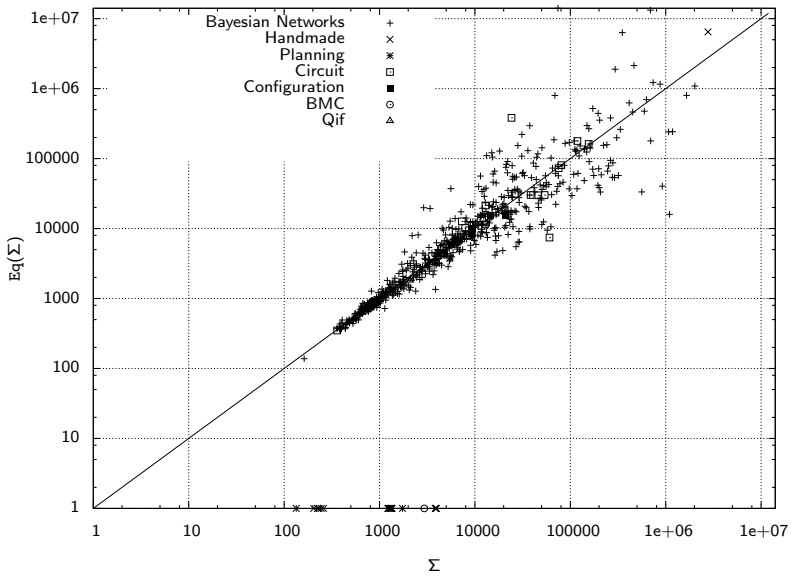

Fig. 50 Comparisons of the sizes of the compiled forms obtained using SDD, when no preprocessing is used vs. when the *eq* combination has been applied first.

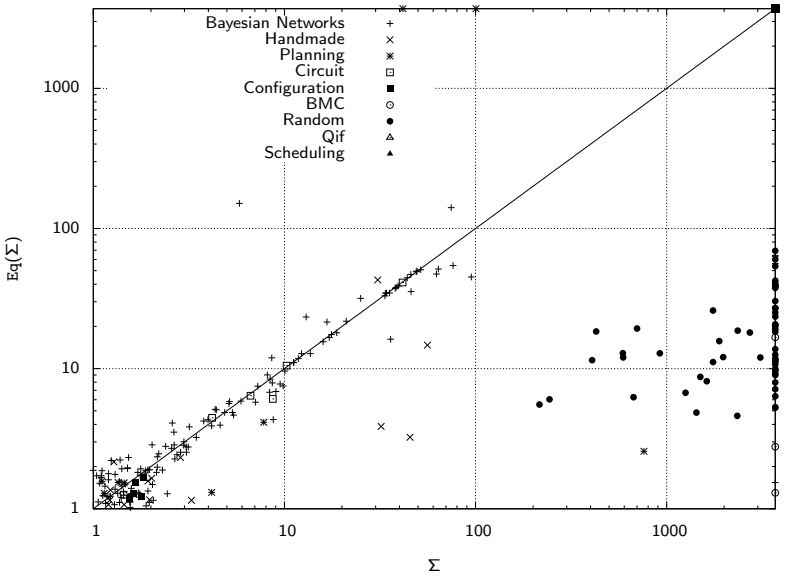

Fig. 51 Comparisons of the compilation times of `cnf2obdd`, when no preprocessing is used vs. when the *eq* combination has been applied first.

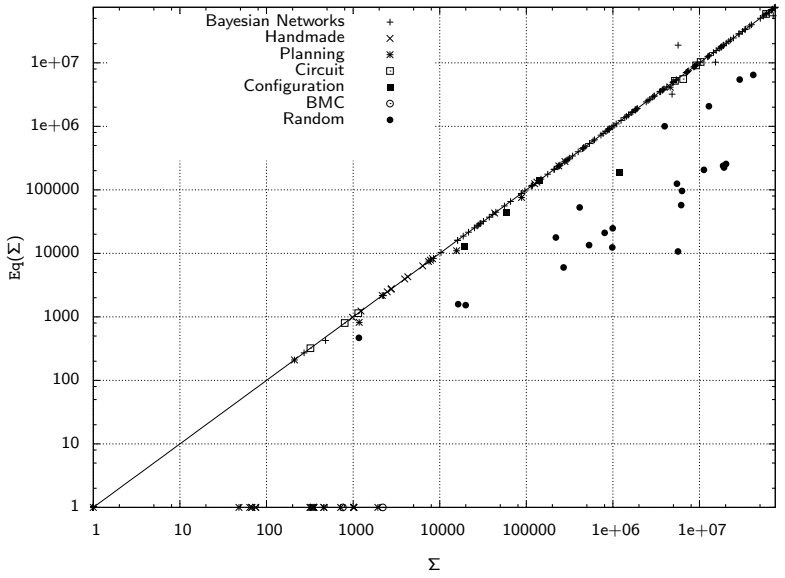

Fig. 52 Comparisons of the sizes of the compiled forms obtained using `cnf2obdd`, when no preprocessing is used vs. when the *eq* combination has been applied first.

5.4 Focusing on the number of instances solved

A	B	#s(A and B)	#s(A or B)	#s(A) - #s(B)	#s(B) - #s(A)
eq+Cachet	Cachet	1083	1113	**29**	1
#*eq*+Cachet	Cachet	1083	1223	**139**	1
#*eq*+Cachet	*eq*+Cachet	1111	1223	**111**	1
eq+sharpSAT	sharpSAT	1092	1138	23	23
#*eq*+sharpSAT	sharpSAT	1099	1243	**128**	16
#*eq*+sharpSAT	*eq*+sharpSAT	1106	1236	**121**	9
eq+SampleCount	SampleCount	1168	1365	**182**	15
#*eq*+SampleCount	SampleCount	1179	1387	**204**	4
#*eq*+SampleCount	*eq*+SampleCount	1336	1397	**47**	14
eq+C2D	C2D	1250	1346	**80**	16
eq+Dsharp	Dsharp	953	990	16	**21**
eq+SDD	SDD	688	857	**127**	42
eq+cnf2obdd	cnf2obdd	259	306	**45**	2

Table 3 Impact of the eq and $\#eq$ preprocessings on the number of instances "solved" for several approaches to model counting.

Tables 3 synthesizes some of the results. Each line compares the performances of two approaches to model counting (say, A and B), using or not eq or $\#eq$. For instance, the first line compares Cachet equipped with the eq preprocessing with Cachet when no preprocessing is performed. #s(A and B) indicates how many instances (over 1449) have been solved by both A and B within the time limit. #s(A or B) indicates how many instances have been solved by A or by B (or by both of them) within the time limit. #s(A) - #s(B) (resp. #s(B) - #s(A)) indicates how many instances have been solved by A but not by B (resp. by B but not by A) within the time limit. In each line, the winner is in bold type, i.e., when a a value in the column $\#s(A) - \#s(B)$ is in bold, it means that $\#s(A) - \#s(B) > \#s(B) - \#s(A)$, so that A has been a better performer than B; similarly, when a a value in the column $\#s(B) - \#s(A)$ is in bold, it means that $\#s(B) - \#s(A) > \#s(A) - \#s(B)$, so that B has been a better performer than A.

The preprocessing time spent by pmc whatever the combination used is systematically included in the times reported in the previous figures and tables. Note that it would not make sense to draw any conclusion about the efficiency of the preprocessing by comparing for each instance the preprocessing time with the overall solving time. Indeed, in some cases, the preprocessing time is (almost) equal to the overall solving time, just because pmc does (almost) all the model counting job!

The empirical results clearly show the efficiency of our preprocessing techniques. Thus, the number of instances which can be solved within the time limit when a preprocessing is used is higher for many downstream model counters, and sometimes significantly higher, than the corresponding number without preprocessing. This happens for all the model counters we tested, except sharpSAT and Dsharp when equipped with the eq preprocessing (which is coherent since Dsharp mainly follows the trace of sharpSAT). A possible explanation is that the use of implicit bcp

in sharpSAT and Dsharp, as well as the (incomplete) backbone detection performed by Dsharp as a preprocessing, make the eq preprocessing useless; the time spent doing it is thus wasted, and this has an influence on the number of instances which can be solved within the time limit.

Interestingly, eq+C2D led to substantial space savings compared to C2D. Especially, our experiments showed that the size of the resulting Decision-DNNF representations can be more than one order of magnitude larger without preprocessing. This can be explained by the treewidth reduction which can be achieved by the eq preprocessing (cf. Figure 26). Indeed, it is known that the only exponential factor in the time complexity of C2D (thus, in the size of the generated Decision-DNNF representation) is the width of the decomposition tree which is computed. Furthermore, this width is bounded by the treewidth of the primal graph of the input CNF instance [14]. The size reduction of the compiled form which results from the eq preprocessing is a strong piece of evidence that its practical impact is not limited to the model counting issue, and that the eq preprocessing also proves useful for equivalence-preserving knowledge compilation.

Finally, contrariwise to eq, the $\#eq$ preprocessing proved useful for every downstream model counter. The increase of the number of instances solved when $\#eq$ is applied is often very significant. One can also observe that the impact of the equivalence/gates detection and replacement is huge (for instance, $\#eq$+Cachet is a much better performer than eq+Cachet).

5.5 Explaining the Impact of eq and $\#eq$ on Model Counting

In order to look for an explanation of the computational benefits offered by the eq combination and the $\#eq$ combination, we performed some additional measurements. Our primary assumptions were that reducing the input CNF instances in term of $\#var$, $\#lit$ and tw would lead to improved performances for subsequent model counting (i.e., a reduction of the computation times needed to count models when direct model counters are used, and a reduction of both the compilation times and the sizes of the compiled representations, when compilation-based model counters are considered). Our further measurements are intended to test these assumptions.

In order to keep the paper in reasonable size limits, only two downstream model counters have been considered: Cachet and C2D.[5] For each of the three measures ($\#var$, $\#lit$ and tw), we drew a dot plot where we reported for Cachet (equipped with $\#eq$) and C2D (equipped with eq), the time improvement as a function of the measure improvement. For eq and C2D, we also added similar dot plots about size improvement. Each time/size/measure improvement has been computed as the ratio between its value without preprocessing minus its value with preprocessing, divided by the value of the measure without preprocessing. This ratio is positive when the preprocessing proved useful, and negative otherwise (which may easily happen – for instance when the instance under consideration has already been preprocessed, preprocessing it again may just be a waste of computational resources when no further

[5] We made similar measurements with the other model counters considered in the paper, and got similar observations. The corresponding results are available from the authors on demand.

reduction is achieved). So as to make significant observations, we selected in the set of 1449 benchmarks only those which are "mildly hard" for model counting (formally, those for which C2D – which was the best performer on our dataset given the huge number of instances encoding Bayesian networks in it – required at least 10s to compile them). The resulting subset of CNF formulae Σ contains 1188 instances for which measure improvements have been computed. Cachet succeeded in counting the number of models for 1082 instances among them within the given time limit. For the tw measure, as already explained, we actually computed a upper bound of $tw(\Sigma)$. We did it using QuickBB (www.hlt.utdallas.edu/~vgogate/quickbb.html) equipped with the min fill heuristic and for an allocated time of 1h. This led us to focus on a proper (and much smaller) subset of the set of instances, containing only 69 CNF formulae (for the remaining ones, QuickBB has not been able to do the job in due time).

Figures 53, 54, and 55 show how the $\#var$ (resp. $\#lit$, tw) improvement obtained by using eq is related to the compilation time improvement obtained by C2D.

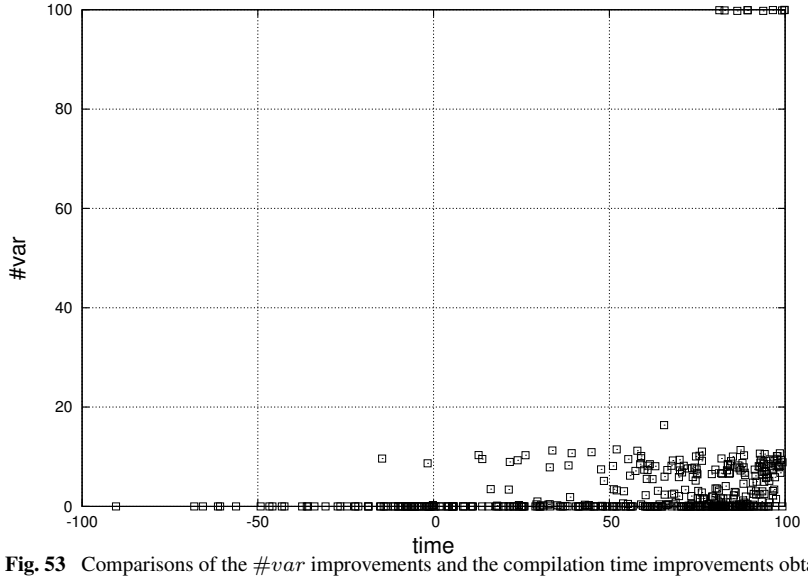

Fig. 53 Comparisons of the $\#var$ improvements and the compilation time improvements obtained when eq and C2D are used.

Figures 56, 57, and 58 report measurements similar to those given on Figures 53, 54, and 55, but focus on compilation size improvements.

Finally, Figures 59, 60, and 61 show how the $\#var$ (resp. $\#lit$, tw) improvement obtained by using $\#eq$ is related to the time improvement for model counting obtained by Cachet.

Looking at Figures 53 to 61, one can first observe that there is no monotonic relationship between any measure improvement and the time (or space) improvement it would lead to, whatever Cachet (equipped with $\#eq$) or C2D (equipped with eq) are used for model counting. This can be easily explained by the fact that some instances are already simplified (or almost simplified) and in this case, as already

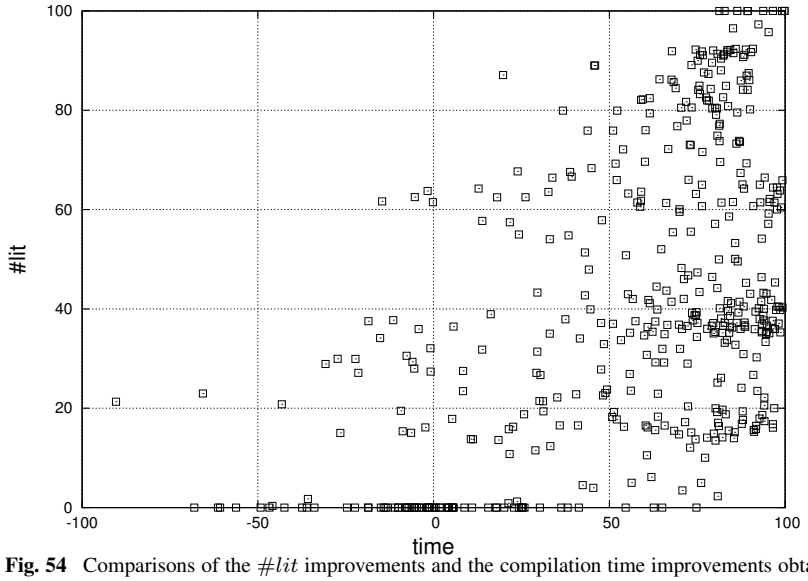

Fig. 54 Comparisons of the $\#lit$ improvements and the compilation time improvements obtained when eq and `C2D` are used.

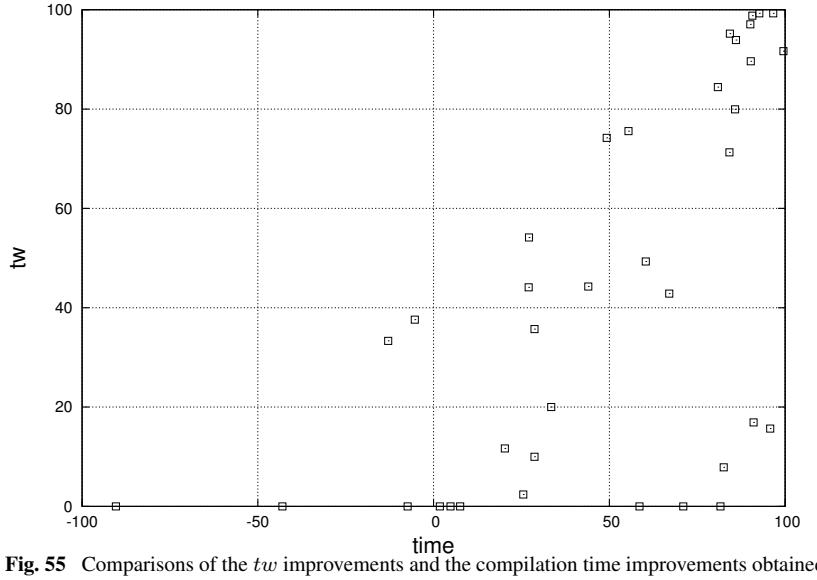

Fig. 55 Comparisons of the tw improvements and the compilation time improvements obtained when eq and `C2D` are used.

mentioned, the preprocessing phase is useless and increases the global solving time. It can even be the case that simplifying the input instance (i.e., reducing its $\#var$, $\#lit$ or tw measure) leads to increasing the corresponding time (solving time for `Cachet` and compilation time for `C2D`), as well as the size of the compiled representation obtained with `C2D`. A reason for it is that the simplification phase may have a strong impact on the branching heuristic (based on VSADS) used by `Cachet` and on the decomposition tree computed by `C2D`. One can observe that such scenarios where

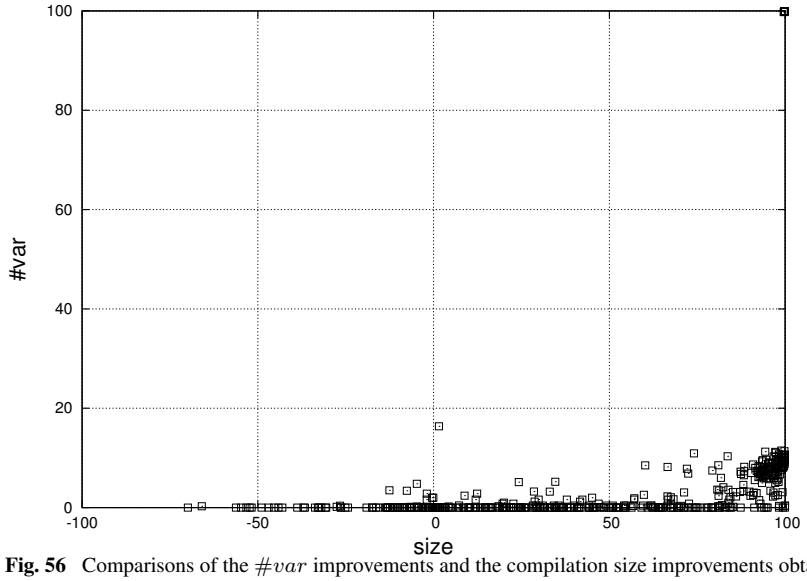

Fig. 56 Comparisons of the $\#var$ improvements and the compilation size improvements obtained when eq and C2D are used.

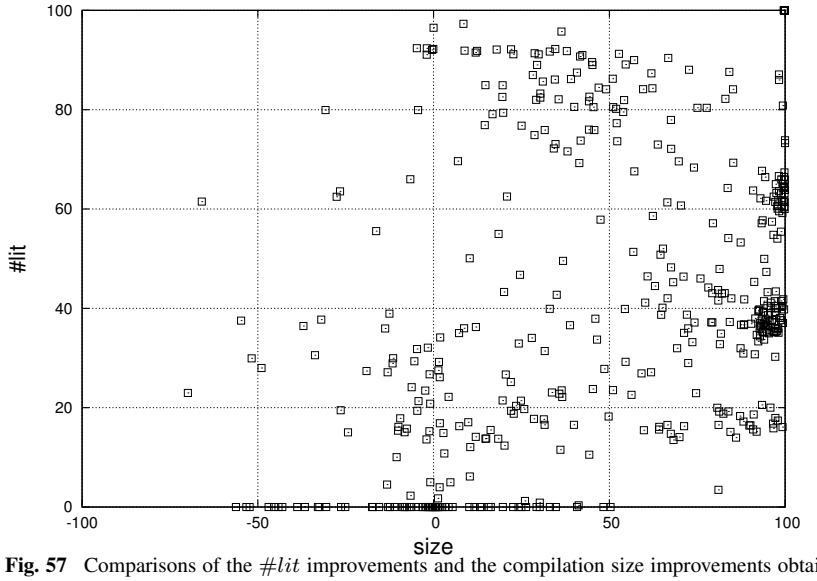

Fig. 57 Comparisons of the $\#lit$ improvements and the compilation size improvements obtained when eq and C2D are used.

preprocessing appears as counterproductive are quite rare when $\#var$ and tw are considered, and more frequent when $\#lit$ is considered.

Another interesting observation is that for a great majority of instances any measure improvement leads to a time (or a space) improvement for the downstream approach to model counting: graphically speaking, the great majority of dots in Figures 53 to 61 are located in the rightmost parts of the figures (those for which the time/space improvements are positive). Of course, this coheres with the results of

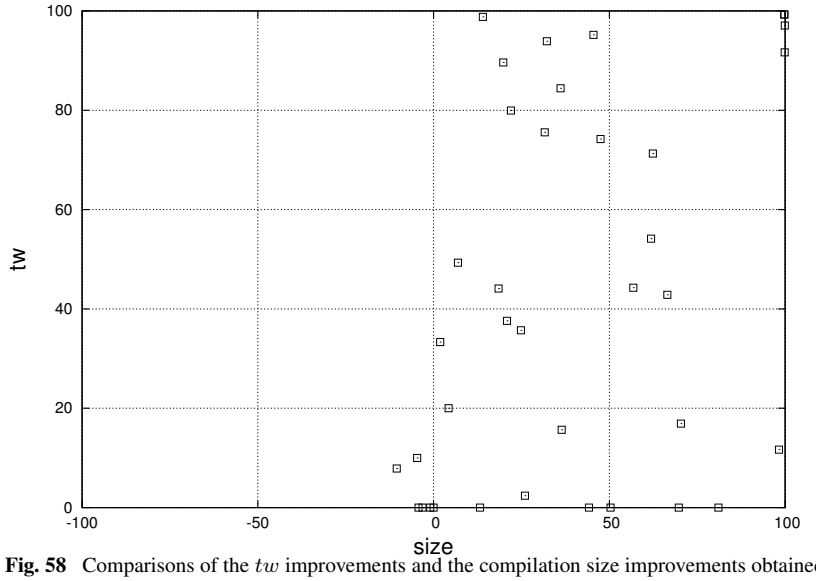

Fig. 58 Comparisons of the tw improvements and the compilation size improvements obtained when eq and `C2D` are used.

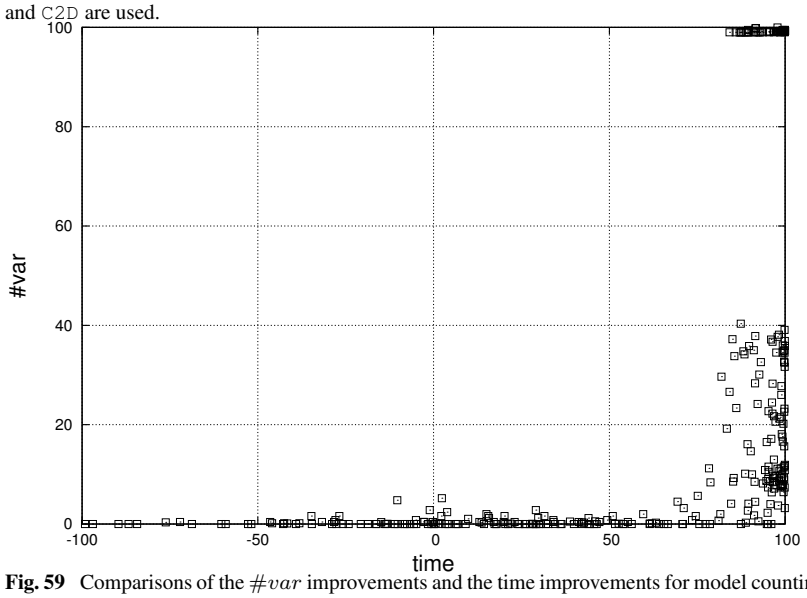

Fig. 59 Comparisons of the $\#var$ improvements and the time improvements for model counting obtained when $\#eq$ and `Cachet` are used.

the experiments discussed previously in the paper and shows that preprocessing is typically useful.

One can also observe that the repartition of the $\#lit$ improvements which have been obtained is more uniform than the repartition of the $\#var$ improvements and that (as expected) $\#eq$ leads to eliminate many more variables than eq. Indeed, most of the time, $\#var$ improvements are of value less than 20% when eq is used, and of value less than 40% when $\#eq$ is used instead. For a few instances, all the variables are eliminated.

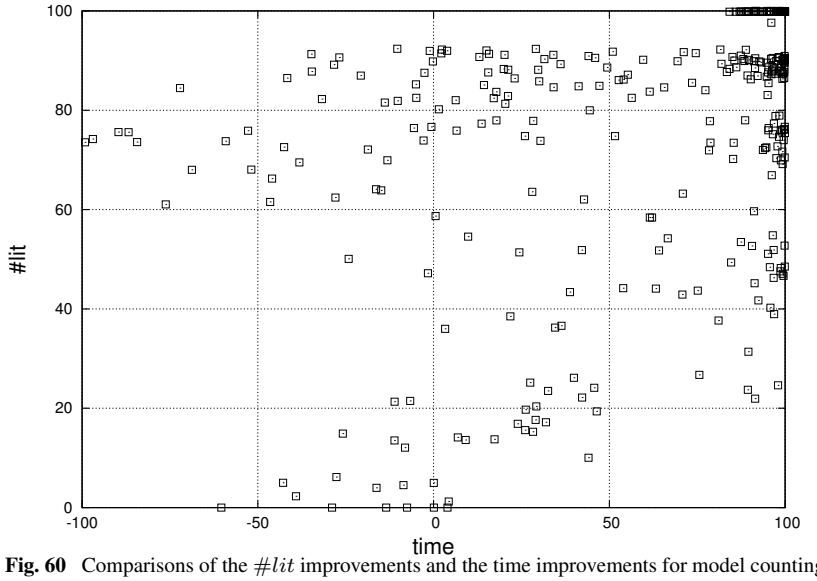

Fig. 60 Comparisons of the $\#lit$ improvements and the time improvements for model counting obtained when $\#eq$ and `Cachet` are used.

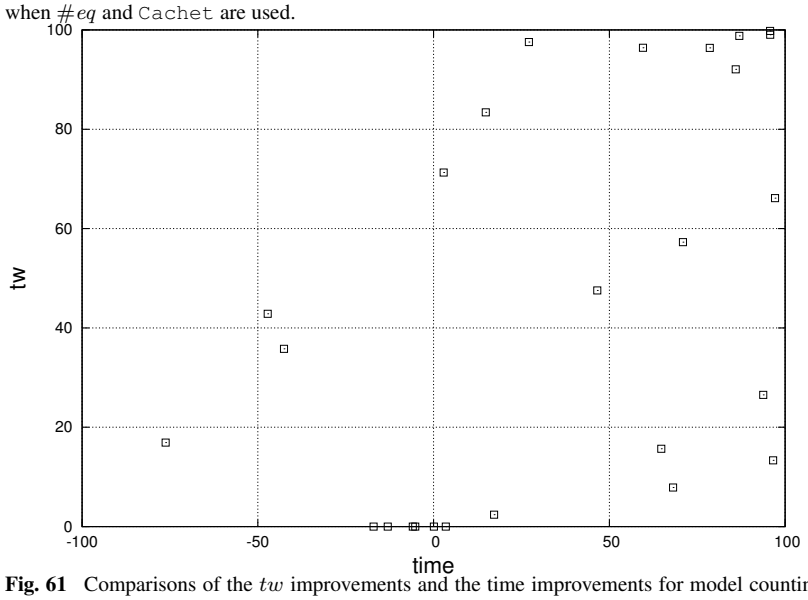

Fig. 61 Comparisons of the tw improvements and the time improvements for model counting obtained when $\#eq$ and `Cachet` are used.

In order to get less informative, yet more readable figures, we also measured mean cumulated improvements for the three measures. For each of $\#var$, $\#lit$ and tw, for each possible improvement ratio r obtained in the experiments for this measure, we gather all instances for which the improvement obtained by preprocessing is $\leq r$. Then we compute the mean value of the improvements for those instances, and we relate it with the mean time/size improvement obtained for the same instances. The resulting dots are then linked (formally, a linear interpolation is achieved) in order to make the curves smooth.

Figure 62 shows how the mean cumulated improvement for $\#var$, $\#lit$ and tw obtained by using $\#eq$ is related to the mean cumulated time improvement for model counting obtained by Cachet.

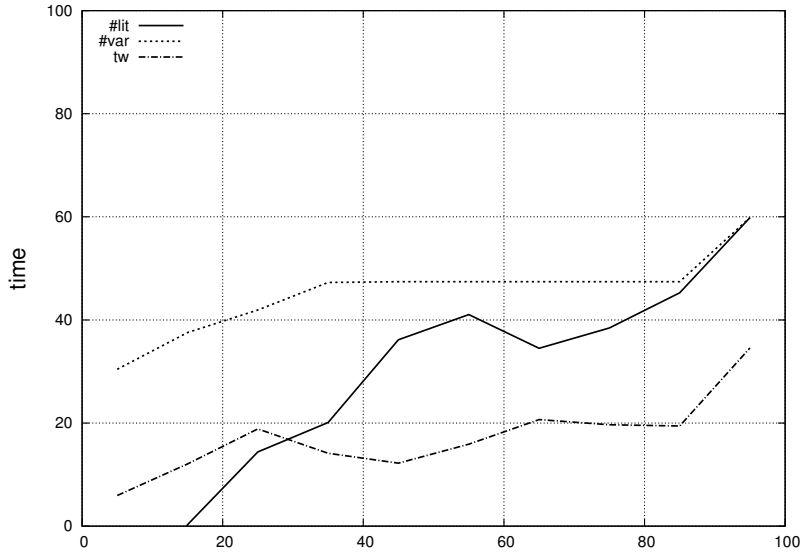

Fig. 62 Comparisons of the mean cumulated improvements for the three measures and the mean cumulated time improvements for model counting obtained when $\#eq$ and Cachet are used.

Figures 63 and 64 show respectively how the mean cumulated improvement for $\#var$, $\#lit$ and tw obtained by using eq is related to the mean cumulated compilation time improvement (respectively the mean cumulated compilation size improvement) obtained by C2D.

One can check on Figures 62, 63 and 64 that none of the reported curves correspond strictly to a monotonic function. However, each of the curves has the shape of a "globally increasing" function, which was what we expected. The "quite flat" aspect of the curves related to $\#var$ is easily explained by that fact that the experiments done led to almost no improvement value in the interval $[20, 100)$ when C2D has been considered (resp. $[40, 80]$ when Cachet has been considered). Finally, it is quite hard to draw any solid conclusion about the tw curves. On the one hand, only upper approximations of the actual tree widths are computed. On the other hand, the curves are based on very few instances.

To sum up, our experiments show slight, yet interesting correlations: from an empirical point of view, reducing the value of $\#var$, $\#lit$ and tw through a preprocessing step often leads to diminish the model counting times (for Cachet), and both the compilation times and the sizes of the compiled representations for C2D.

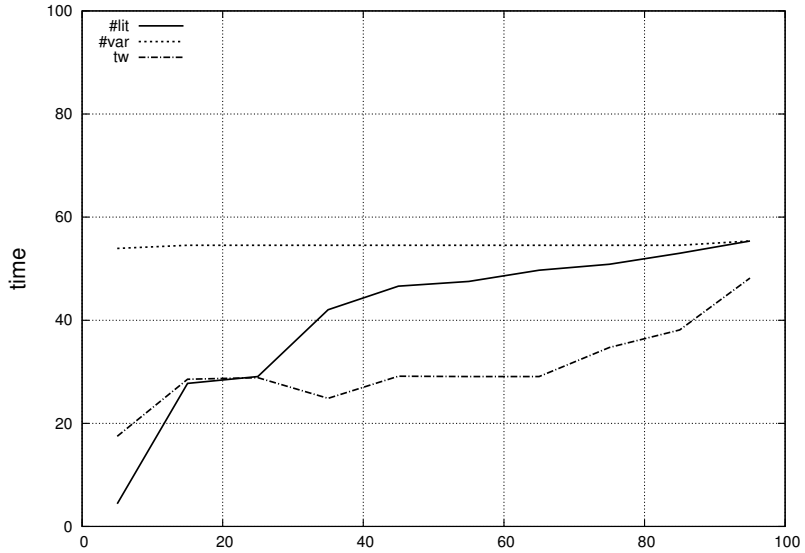

Fig. 63 Comparisons of the mean cumulated improvements for the three measures and the mean cumulated compilation time improvements obtained when *eq* and C2D are used.

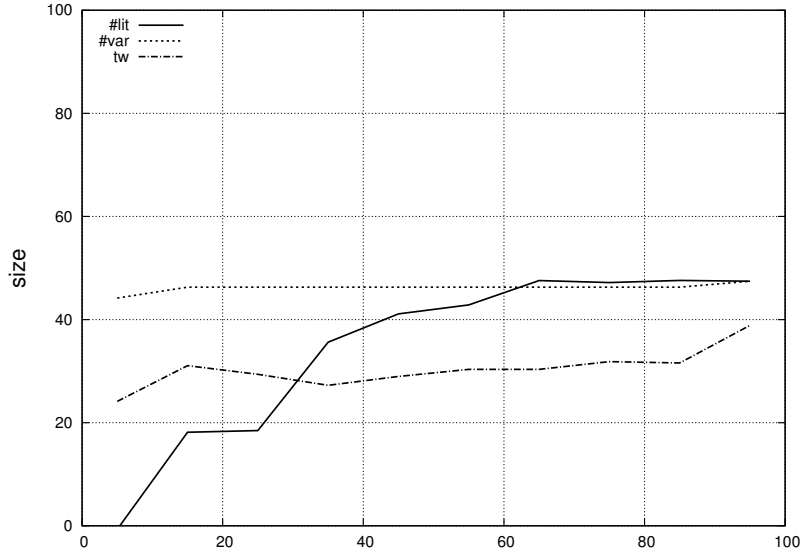

Fig. 64 Comparisons of the mean cumulated improvements for the three measures and the mean cumulated compilation size improvements obtained when *eq* and C2D are used.

6 Other Related Work

As evoked in the introduction, many preprocessing techniques have been considered so far for SAT solving and QBF solving, see e.g., [27,30]. Some of them do not preserve the number of models of the input, and we cannot take advantage of such techniques given our model counting objective. Among the remaining preprocessings are the following equivalence-preserving techniques:

- Failed Literal Elimination (FLE) [23],
- Self-Subsuming Resolution (SSR) [21],
- Hidden Literal Elimination (HLE) [29],
- Subsumption Elimination (SE) [12,37,7,21],
- Hidden Subsumption Elimination (HSE) [27],
- Asymmetric Subsumption Elimination (ASE) [27,30],
- Tautology Elimination (TE) [7,21],
- Hidden Tautology Elimination (HTE) [27],
- Asymmetric Tautology Elimination (ATE) [27,30].

Those simplification techniques can be gathered into three categories: the first one consists of clause addition techniques which lead to add some unit clauses to the input CNF formula Σ (FLE belongs to it), the second one consists of the clause reduction techniques which lead to remove some literals in clauses of Σ (SSR and HLE belong to it), and the third one consists of the clause elimination techniques which lead to remove clauses from Σ (SE, HSE, ASE, TE, HTE and ATE belong to it).

Indeed, FLE consists in adding to the input CNF formula Σ the complements of the failed literals in it, i.e., those literals ℓ such that $\texttt{bcp}(\Sigma \cup \{\sim\ell\}) = \{\emptyset\}$. SSR consists in replacing every clause α of the input CNF formula Σ which has a resolvent with another clause of Σ, such that this resolvent subsumes α, by this resolvent. HLE consists in exploiting the implication graph associated with the set of binary clauses occurring in Σ. Basically, given a clause α of Σ such that α contains a literal ℓ, the set $HL(\Sigma, \ell)$ containing the complements of all literals reachable from $\sim\ell$ in the binary implication graph, is computed. Then α is simplified by removing from it every literal belonging to $HL(\Sigma, \ell)$. SE (resp. TE) simply consists in removing from the input CNF formula Σ the clauses which are properly subsumed (resp. which are valid). The remaining techniques listed above also consist in removing some clauses from the input when they are detected as subsumed or as tautologies, after applying a literal addition rule to them. This rule is referred to as the Hidden Literal Addition rule (HLA) for HSE and HTE, and as the Asymmetric Literal Addition rule (ALA) for ASE and ATE. Given a CNF formula Σ containing a clause $\ell_1 \vee \ldots \vee \ell_k \vee \ell$ and another clause α of Σ containing the literals $\ell_1, \ldots, \ell_k$, applying the ALA rule to α given Σ consists in replacing α by $\alpha \vee \sim\ell$. The HLA rule is the restriction of the ALA rule to the case $k = 1$.

It is easy to check that `backboneSimpl` is a clause addition technique which is more effective than FLE in the sense that every unit clause added using FLE is also obtained using `backboneSimpl`, but the converse does not hold. Furthermore, in [27] it is shown that ATE is more effective (or powerful) than any preprocessing technique among the remaining clause elimination techniques (SE, TE, HSE, HTE, ASE) in the sense that every clause which can be removed using any of those techniques can be removed as well using ATE, but the converse does not hold.

We now show that, on the one hand, `occurrenceSimpl` is more efficient w.r.t. clause reduction than SSR and HLE, and on the other hand, that `vivificationSimpl` is more efficient than ATE w.r.t. clause elimination.

Proposition 1 *If a literal ℓ of a clause α from Σ is eliminated using SSR or HLE, then ℓ is eliminated from α as well using* `occurrenceSimpl`.

Proof

- SSR: ℓ can be removed from α using SSR iff there exists a clause $\sim\ell \vee \beta$ in Σ such that every literal from β is a literal from $\alpha \setminus \{\ell\}$. Hence, every literal from $\sim\ell \vee \beta$ is the complement of a literal from $\{\ell\} \cup \{\sim(\alpha \setminus \{\ell\})\}$. Accordingly, $\texttt{bcp}(\Sigma \cup \{\ell\} \cup \{\sim(\alpha \setminus \{\ell\})\}) = \{\emptyset\}$, which implies that ℓ can be removed from α using `occurrenceSimpl`.
- HLE: Let ℓ be a literal from a clause α of Σ. Suppose that α contains also a literal ℓ' such that $\ell \in HL(\Sigma, \ell')$. Then there exists a subset B of Σ consisting of binary clauses $\ell' \vee \ell_1, \sim\ell_1 \vee \ell_2, \ldots, \sim\ell_k \vee \sim\ell$. We have that $\ell' \in \alpha \setminus \{\ell\}$. Hence $\sim\ell' \in \sim(\alpha \setminus \{\ell\})\})$. Now, from B and ℓ, by unit propagation the literal ℓ' is generated. Accordingly, since $\sim\ell' \in \sim(\alpha \setminus \{\ell\})\})$, we obtain that $\texttt{bcp}(\Sigma \cup \{\ell\} \cup \{\sim(\alpha \setminus \{\ell\})\}) = \{\emptyset\}$. This shows that ℓ can be removed from α using `occurrenceSimpl`.

Proposition 2 *If a clause α from Σ is eliminated using ATE, then α is eliminated as well using* `vivificationSimpl`.

Proof Let $ALA(\Sigma, \alpha)$ be the set of literals ℓ which can be added to a clause α of Σ via the application of the ALA rule. We show that for each $\ell \in ALA(\Sigma, \alpha)$, in the implication graph associated with $\texttt{bcp}((\Sigma \setminus \{\alpha\}) \cup \{\sim\alpha\})$ there exists a node labelled with $\sim\ell$. The proof is by induction on the number n of applications of the ALA rule needed to generate the whole set $ALA(\Sigma, \alpha)$.

Let $ALA(\Sigma, \alpha)_m$ $(0 \leq m \leq n)$ be the set consisting of the m literals obtained by applying the ALA rule to α given Σ m times successively. Let α_m $(0 \leq m \leq n)$ be the clause obtained by adding to $\alpha_0 = \alpha$ all literals from $ALA(\Sigma, \alpha)_m$.

The base case is when $n = 1$. Then there exists a clause $\ell_1 \vee \ldots \vee \ell_k \vee \ell$ in $\Sigma \setminus \{\alpha_0\}$ such that $\ell_i \in \alpha_0$ $(i \in 1, \ldots, k)$, leading to replace α_0 by $\alpha_1 = \alpha_0 \vee \sim\ell$. The literal $\sim\ell$ is added to $ALA(\Sigma, \alpha)_0 = \emptyset$, leading to $ALA(\Sigma, \alpha)_1 = \{\sim\ell\}$. Since $\ell_i \in \alpha$ $(i \in 1, \ldots, k)$ and $\ell_1 \vee \ldots \vee \ell_k \vee \ell$ belongs to $\Sigma \setminus \{\alpha\}$, it is easy to verify that ℓ can be derived by unit propagation from $(\Sigma \setminus \{\alpha\}) \cup \{\sim\alpha\}$.

Now, consider the general case $n > 1$. By induction hypothesis, suppose that every ℓ generated by applying the ALA rule to α_{m-1} (with $m - 1 < n$) given Σ, there exists a node in the implication graph associated with $\texttt{bcp}((\Sigma \setminus \{\alpha\}) \cup \{\sim\alpha\})$ which is labelled with $\sim\ell$. We have to show that the literal $\sim\ell$ generated by applying the ALA rule to α_m given Σ satisfies the same property. We know that there exists a clause $\ell_1 \vee \ldots \vee \ell_k \vee \ell$ in $\Sigma \setminus \{\alpha_m\}$ such that $\ell_i \in \alpha \cup ALA(\Sigma, \alpha)_m$ $(i \in 1, \ldots, k)$, leading to replace α_m by $\alpha_{m+1} = \alpha_m \vee \sim\ell$ and to add $\sim\ell$ to $ALA(\Sigma, \alpha)_m$ in order to produce $ALA(\Sigma, \alpha)_{m+1}$. Let us consider any ℓ_i $(i \in 1, \ldots, k)$. If $\ell_i \in \alpha$, then by construction $\sim\ell_i \in \sim\alpha$, hence $\sim\ell_i$ belongs to the implication graph associated with $\texttt{bcp}((\Sigma \setminus \{\alpha\}) \cup \{\sim\alpha\})$. If $\ell_i \in ALA(\Sigma, \alpha)_m$, then by induction hypothesis $\sim\ell_i$ also belongs to this graph.

Finally, let α be any clause from Σ such that α is removed using the ATE rule. This implies that the clause α_n is valid. Since for every literal ℓ of α_n, ℓ belongs to

$\alpha \cup ALA(\Sigma, \alpha)_n$, we obtain that every literal $\sim\ell$ belongs to the implication graph associated with $\texttt{bcp}((\Sigma \setminus \{\alpha\}) \cup \{\sim\alpha\})$. Since α_n is valid, there exist two complementary literals in this graph, implying that $\texttt{bcp}((\Sigma \setminus \{\alpha\}) \cup \{\sim\alpha\}) = \{\emptyset\}$. In this case, by definition of `vivificationSimpl`, α is removed by `vivificationSimpl`.

In the main loop of `pmc` equipped with the *eq* combination or the $\#eq$ combination, `occurrenceSimpl` is performed first in order to possibly get more propagation power for `bcp`, and `vivificationSimpl` is performed last in order to remove useless clauses. Together with the previous effectiveness results recalled above, Propositions 1 and 2 explain why none of the nine equivalence-preserving techniques considered here has been taken into account explicitly in any of the two combinations.[6] Using in addition any of the other techniques described above would not lead to get more effective clause reductions or eliminations.

To sum up, while the other techniques may prove useful in practice when the satisfiability issue is targeted, it turns out that considering more efficient (yet more time demanding) preprocessings such as `occurrenceSimpl` and `vivificationSimpl` makes sense when the objective is more time consuming, as it is the case for model counting. Of course, it cannot be excluded that considering less effective preprocessing techniques (in term of clause reduction/clause elimination) would lead to diminish the preprocessing time, hence the overall solving time, and further experiments would be needed to determine whether this is the case. Nevertheless, the simplification times about the preprocessing combinations we implemented, as reported in Section 5.1, are typically very small. In addition, for the elementary preprocessing techniques we have considered in our experiments, the more effective the technique (in term of clause reduction/clause elimination) the more significant the reduction of the overall solving time. This is particularly salient for the gate detection and replacement preprocessings, as described in Section 5.4.

7 Conclusion

We have implemented a preprocessor `pmc` for model counting which includes several preprocessing techniques, especially vivification, occurrence reduction, backbone identification, as well as equivalence, AND and XOR gate identification and replacement.

The experimental results we have obtained show that `pmc` is useful for many downstream model counters. Especially, it often leads to significantly decrease the time needed for model counting when direct model counters are used downstream, and the size of the compiled forms when compilation-based model counters are considered.

If the main contribution of the paper consists in presenting a preprocessor for model counting, and in proving it practically efficient, some other contributions of different nature are also worth being noted. From an algorithmic point of view, our

[6] Similarly, it is easy to check that the Equivalent Literal Substitution (ELS) technique which consists in computing first the strongly connected components of the binary implication graph of Σ, then in replacing in Σ every literal by a representative of its equivalence class is less effective than `equivSimpl`.

technique for detecting AND/OR gates turns out to be original and efficient. In particular, it is somehow "semantical" (i.e., based on BCP) and not "syntactical" (i.e., based on pattern matching). From a more theory-oriented side, we have shown that replacing equivalent literals in a CNF formula can increase the tree width of the associated primal graph. We have also obtained a number of results concerning the confluence, the projectivity and the relative efficiency of some preprocessing techniques.

This work opens several perspectives for further research. Finding other efficient preprocessings and incorporating them into pmc, and testing some additional combinations are some of them. Especially, detecting and replacing If-Then-Else gates (as considered in [21]) could prove useful. It would also be interesting to determine the "best" combinations of elementary preprocessings, depending on the benchmark families. Assessing whether pmc proves useful upstream to other knowledge compilation techniques (for instance, those described in [10,53,22,9]) would be valuable as well.

As future work, we plan also to investigate whether the preprocessing techniques we have considered could be adapted to the inprocessing case [33] (i.e., being used within a "direct" model counter at some decision nodes).

Exploiting the preprocessing techniques based on equivalence or gate detection and replacement in the case a compilation-based approach is used downstream, has to be investigated. Indeed, if $\Sigma \models l \leftrightarrow \Phi$ holds, then replacing Σ by $\Sigma[\ell \leftarrow \Phi] \wedge (\ell \leftrightarrow \Phi)$ is an equivalence-preserving transformation. When the set PS of propositional variables can be partitioned into two sets, a set C of controllable variables and a set U of uncontrollable variables, so that conditioning the variables of U is never required, such an equivalence-preserving transformation can prove useful in a compilation-based approach to model counting. The idea is to look for equivalences of the form $\ell \leftrightarrow \Phi$ when $var(\ell) \in U$, only. Indeed, given a consistent term γ built up from variables of C, only, the conditioning $\Sigma \mid \gamma$ of Σ by γ is equivalent to $(\Sigma[\ell \leftarrow \Phi] \mid \gamma) \wedge (\ell \leftrightarrow (\Phi \mid \gamma))$, hence the number of models of $\Sigma \mid \gamma$ is equal to the number of models of $(\Sigma[\ell \leftarrow \Phi] \mid \gamma)$. Compiling $\Sigma[\ell \leftarrow \Phi]$ into a representation from which both the conditioning transformation and the model counting query are tractable thus prove enough to be able to compute the number of models of Σ conditioned by any term built up from variables of C. Interestingly, problems for which a restricted form of conditioning is sufficient occur in the model-based diagnosis area, and in the planning area. It would be interesting to determine empirically whether such detections would prove computationally helpful.

Finally, another interesting issue to be investigated concerns the impact of pmc on *model enumeration*, i.e., the problem which consists in listing successively all the models of a given propositional formula Σ. Obviously, model enumeration is computationally hard when no restriction is imposed on Σ, because the number of models of Σ can be exponential in the size of Σ (and there is nothing to do for circumventing this source of complexity), but also because generating an additional model (or determining that all the models have been generated) also is computationally demanding. Clearly enough, pmc equipped with the *eq* combination can be safely used to simplify any Σ before enumerating its models, and it would be useful to characterize families of instances Σ for which this is computationally helpful. Furthermore, provided that the definitions $\ell_1 \leftrightarrow \Phi_1, \ldots, \ell_k \leftrightarrow \Phi_k$ are stored when

they are found, gate detection and replacement can also be exploited for model enumeration. Indeed, by construction, there exists a one-to-one mapping between the set of models of Σ over $Var(\Sigma)$ and the set of models of $\Sigma[\ell_1 \leftarrow \Phi_1, \ldots, \ell_k \leftarrow \Phi_k]$ (i.e., the formula obtained by replacing in Σ every gate found by its definition) over $Var(\Sigma[\ell_1 \leftarrow \Phi_1, \ldots, \ell_k \leftarrow \Phi_k])$. Given any model $\mathcal{I}_r$ of $\Sigma[\ell_1 \leftarrow \Phi_1, \ldots, \ell_k \leftarrow \Phi_k]$ over its set of variables, determining the corresponding model $\mathcal{I}$ of Σ over $Var(\Sigma)$ simply consists in determining the truth value of each $var(\ell_i)$ ($i \in 1, \ldots, k$) in $\mathcal{I}$ (since $\mathcal{I}_r$ is a restriction of $\mathcal{I}$). But the truth value of each $var(\ell_i)$ ($i \in 1, \ldots, k$) in $\mathcal{I}$ precisely is the truth value in $\mathcal{I}_r$ of the associated definition Φ_i. Hence completing $\mathcal{I}_r$ to generate the corresponding $\mathcal{I}$ is not computationally demanding when the definitions $\ell_1 \leftrightarrow \Phi_1, \ldots, \ell_k \leftrightarrow \Phi_k$ have been computed first. Making some further experiments to determine the extent to which pmc equipped with the $\#eq$ combination is useful for model enumeration would thus be of interest. This could be particularly valuable when compilation-based model counters are considered. Indeed, when the input formula has been first turned into a compiled form, model enumeration can be typically achieved with a polynomial-delay algorithm (an enumeration algorithm has polynomial delay if the elapsed time between two successive outputs is polynomial in the size of the input).

Acknowledgements This work has been partially supported by the project BR4CP ANR-11-BS02-008 of the French National Agency for Research.

References

1. F.A. Aloul, I.L. Markov, and K.A. Sakallah. MINCE: A static global variable-ordering heuristic for SAT search and BDD manipulation. *J. UCS*, 10(12):1562–1596, 2004.
2. U. Apsel and R. I. Brafman. Lifted MEU by weighted model counting. In *Proc. of AAAI'12*, 2012.
3. G. Audemard, J.-M. Lagniez, and L. Simon. Just-in-time compilation of knowledge bases. In *Proc. of IJCAI'13*, pages 447–453, 2013.
4. G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solver. In *Proc. of IJCAI'09*, pages 399–404, 2009.
5. F. Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In *Proc. of SAT'04*, pages 341–355, 2004.
6. A. Biere. Lingeling essentials, A tutorial on design and implementation aspects of the the SAT solver lingeling. In *Proc. of POS'14. Fifth Pragmatics of SAT workshop, a workshop of the SAT 2014 conference, part of FLoC 2014 during the Vienna Summer of Logic*, page 88, 2014.
7. A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.
8. L. Bordeaux, M. Janota, J. Marques-Silva, and P. Marquis. On unit-refutation complete formulae with existentially quantified variables. In *Proc. of KR'12*, pages 75–84, 2012.
9. L. Bordeaux and J. Marques-Silva. Knowledge compilation with empowerment. In *Proc. of SOFSEM'12*, pages 612–624, 2012.
10. Y. Boufkhad, E. Grégoire, P. Marquis, B. Mazure, and L. Saïs. Tractable cover compilations. In *Proc. of IJCAI'97*, pages 122–127, 1997.
11. Y. Boufkhad and O. Roussel. Redundancy in random SAT formulas. In *Proc. of AAAI'00*, pages 273–278, 2000.
12. C.L. Chang and R.C.T. Lee. *Symbolic logic and mechanical theorem proving*. Academic Press, New York, 1973.
13. M. Chavira and A. Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6-7):772–799, 2008.
14. A. Darwiche. Decomposable negation normal form. *Journal of the ACM*, 48(4):608–647, 2001.

15. A. Darwiche. New advances in compiling cnf into decomposable negation normal form. In *Proc. of ECAI'04*, pages 328–332, 2004.
16. A. Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *Proc. of IJCAI'11*, pages 819–826, 2011.
17. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Journal of the ACM*, 5(7):394–397, July 1962.
18. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.
19. A. del Val. Tractable databases: How to make propositional unit resolution complete through compilation. In *Proc. of KR'94*, pages 551–561, 1994.
20. C. Domshlak and J. Hoffmann. Fast probabilistic planning through weighted model counting. In *Proc. of ICAPS'06*, pages 243–252, 2006.
21. N. Een and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proc. of SAT'05*, pages 61–75, 2005.
22. H. Fargier and P. Marquis. Extending the knowledge compilation map: Krom, Horn, affine and beyond. In *Proc. of AAAI'08*, pages 442–447, 2008.
23. J. W. Freemann. *Improvement to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, 1995.
24. A. Van Gelder. Toward leaner binary-clause reasoning in a satisfiability solver. *Annals of Mathematics and Artificial Intelligence*, 43(1):239–253, 2005.
25. C. P. Gomes, J. Hoffmann, A. Sabharwal, and B. Selman. From sampling to model counting. In *Proc. of IJCAI'07*, pages 2293–2299, 2007.
26. H. Han and F. Somenzi. Alembic: An efficient algorithm for CNF preprocessing. In *Proc. of DAC'07*, pages 582–587, 2007.
27. M. Heule, M. Järvisalo, and A. Biere. Clause elimination procedures for CNF formulas. In *Proc. of LPAR'10*, pages 357–371, 2010.
28. M. Heule, M. Järvisalo, and A. Biere. Covered clause elimination. In *Proc. of LPAR'10*, pages 41–46, 2010.
29. M. Heule, M. Järvisalo, and A. Biere. Efficient cnf simplification based on binary implication graphs. In *Proc. of SAT'11*, pages 201–215, 2011.
30. 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.
31. J. Huang and A. Darwiche. Using DPLL for efficient OBDD construction. In *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*, pages 157–172, 2004.
32. M. Järvisalo, A. Biere, and M. Heule. Simulating circuit-level simplifications on CNF. *Journal of Automated Reasoning*, 49(4):583–619, 2012.
33. M. Järvisalo, M. Heule, and A. Biere. Inprocessing rules. In *Proc. of IJCAR'12*, pages 355–370, 2012.
34. R. J. Bayardo Jr. and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proc. of AAAI'97*, pages 203–208, 1997.
35. O. Kullmann. On a generalization of extended resolution. *Discrete Applied Mathematics*, 96-97:149–176, 1999.
36. P. Liberatore. Redundancy in logic I: CNF propositional formulae. *Artificial Intelligence*, 163(2):203–232, 2005.
37. D.W. Loveland. *Automated theorem proving: a logical basis*. Noth-Holland, Amsterdam, 1978.
38. I. Lynce and J. Marques-Silva. Probing-based preprocessing techniques for propositional satisfiability. In *Proc. ICTAI'03*, pages 105–110, 2003.
39. N. Manthey. Coprocessor 2.0 - A flexible CNF simplifier - (tool presentation). In *Proc. of SAT'12*, pages 436–441, 2012.
40. N. Manthey. Solver description of RISS 2.0 and PRISS 2.0. Technical report, TU Dresden, Knowledge Representation and Reasoning, 2012.
41. R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity from characteristic 'phase transitions'. *Nature*, 33:133–137, 1999.
42. M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. of DAC'01*, pages 530–535, 2001.
43. Ch.J. Muise, Sh.A. McIlraith, J.Ch. Beck, and E.I. Hsu. Dsharp: Fast d-DNNF compilation with sharpSAT. In *Proc. of AI'12*, pages 356–361, 2012.

44. R. Ostrowski, É. Grégoire, B. Mazure, and L. Saïs. Recovering and exploiting structural knowledge from CNF formulas. In *Proc. of CP'02*, pages 185–199, 2002.
45. H. Palacios, B. Bonet, A. Darwiche, and H. Geffner. Pruning conformant plans by counting models on compiled d-DNNF representations. In *Proc. of ICAPS'05*, pages 141–150, 2005.
46. C. Piette, Y. Hamadi, and L. Saïs. Vivifying propositional clausal formulae. In *Proc. of ECAI'08*, pages 525–529, 2008.
47. N. Robertson and P.D. Seymour. Graph minors. III. planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984.
48. D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1–2):273–302, 1996.
49. M. Samer and S. Szeider. Algorithms for propositional model counting. *Journal of Discrete Algorithms*, 8(1):50–64, 2010.
50. T. Sang, F. Bacchus, P. Beame, H.A. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *Proc. of SAT'04*, 2004.
51. T. Sang, P. Beame, and H. A. Kautz. Performing Bayesian inference by weighted model counting. In *Proc. of AAAI'05*, pages 475–482, 2005.
52. J. P. Marques Silva and K. A. Sakallah. GRASP - a new search algorithm for satisfiability. In *Proc. of ICCAD'96*, pages 220–227, 1996.
53. S. Subbarayan, L. Bordeaux, and Y. Hamadi. Knowledge compilation properties of tree-of-BDDs. In *Proc. of AAAI'07*, pages 502–507, 2007.
54. S. Subbarayan and D.K. Pradhan. NiVER: Non increasing variable elimination resolution for preprocessing SAT instances. In *Proc. of SAT'04*, pages 276–291, 2004.
55. M. Thurley. sharpSAT - counting models with advanced component caching and implicit BCP. In *Proc. of SAT'06*, pages 424–429, 2006.
56. T. Toda and K. Tsuda. BDD construction for all solutions SAT and efficient caching mechanism. In *Proc. of SAC'15, Track on Constraint Solving and Programming and Knowledge Representation and Reasoning (CSP-KR)*, 2015.
57. L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.
58. H. Zhang and M.E. Stickel. An efficient algorithm for unit propagation. In *Proc. of ISAIM96*, pages 166–169, 1996.
59. L. Zhang, C.F. Madigan, M.W. Moskewicz, and S. Malik. Efficient conflict driven learning in Boolean satisfiability solver. In *Proc. of ICCAD'01*, pages 279–285, 2001.