Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence

Preprocessing for Propositional Model Counting

Jean-Marie Lagniez and **Pierre Marquis**
CRIL-CNRS and Université d'Artois, Lens, France
{lagniez, marquis}@cril.fr

Abstract

This paper is concerned with preprocessing techniques for propositional model counting. We have implemented a preprocessor which includes many elementary preprocessing techniques, including occurrence reduction, vivification, backbone identification, as well as equivalence, AND and XOR gate identification and replacement. We performed intensive experiments, using a huge number of benchmarks coming from a large number of families. Two approaches to model counting have been considered downstream: "direct" model counting using `Cachet` and compilation-based model counting, based on the `C2D` compiler. The experimental results we have obtained show that our preprocessor is both efficient and robust.

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 (of course, the preprocessing time is taken into account in the global solving time). Some preprocessing techniques are nowadays acknowledged as valuable for SAT solving (see (Bacchus and Winter 2004; Subbarayan and Pradhan 2004; Lynce and Marques-Silva 2003; Een and Biere 2005; Piette, Hamadi, and Saïs 2008; Han and Somenzi 2007; Heule, Järvisalo, and Biere 2010; Järvisalo, Biere, and Heule 2012; Heule, Järvisalo, and Biere 2011)), leading to computational improvements. As such, they are now embodied in many state-of-the-art SAT solvers, like `Glucose` (Audemard and Simon 2009) which takes advantage of the `Satellite` preprocessor (Een and Biere 2005).

In this paper, we focus on preprocessing techniques p for *propositional model counting*, i.e., the problem which consists in determining the number of truth assignments satisfying a given propositional formula Σ. Model counting and its direct generalization, weighted model counting,[1] are central to many AI problems including probabilistic inference (see e.g., (Sang, Beame, and Kautz 2005; Chavira and Darwiche 2008; Apsel and Brafman 2012)) and forms of planning (see e.g., (Palacios et al. 2005; Domshlak and Hoffmann 2006)). However, model counting is a computationally demanding task (it is #P-complete (Valiant 1979) 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 $\epsilon > 0$ (Roth 1996)). Especially, it is harder (both in theory and in practice) than SAT.

Focussing 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 p considered for SAT must be let aside; this includes the *pure literal rule* (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* (replacing in the input CNF formula all the clauses containing a given variable x by the set of all their resolvents over x) or the *blocked clause elimination rule* (removing every clause containing a literal such that every resolvent obtained by resolving on it is a valid clause). Indeed, these preprocessings preserve only the satisfiability of the input formula but not its number of models. On the other hand, the high complexity of model counting allows for considering more aggressive, 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; contrastingly, 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. Thus it would not make sense to consider backbone detection as a preprocessing for SAT.

Another important aspect for the choice of candidate preprocessing techniques p 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.,

[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.

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 using a compilation-based approach, the time used to compile is not as significant as the size of the compiled form (as soon as it can be balanced by sufficiently many on-line queries). Hence it is natural to focus on the impact of p 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: vivification, occurrence reduction, backbone identification, as well as equivalence, AND and XOR gate identification and replacement. The three former techniques preserve equivalence, 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. 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 a large number of families. We focussed on two combinations of elementary preprocessing techniques (one of them preserves equivalence, and the other one preserves the number of models only). We considered two model counters, the "direct" model counter `Cachet` (Sang et al. 2004), as well as a compilation-based model counter, based on the `C2D` compiler targeting the `d-DNNF` language, consisting of DAG-based representations in deterministic, decomposable negation normal form (Darwiche 2001). The experimental results we have obtained show that the two combinations of preprocessing techniques for model counting we have focussed on are useful for model counting. Significant time (or space savings) can be obtained, when taking advantage of them. Unsurprinsingly, the level of improvements which can be achieved typically depends on the family of the instance Σ and on the preprocessings used. Nevertheless, each of the two combinations of elementary preprocessing techniques we have considered appears as robust from the experimental standpoint.

The rest of the paper is organized as follows. The next section gives some formal preliminaries. Then the elementary preprocessings we have considered are successively presented. Afterwards, some empirical results are presented, showing the usefulness of the preprocessing techniques we took advantage of for the model counting issue. Finally, the last section concludes the paper and presents some perspectives for further research.

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.

$PROP_{PS}$ 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.

We also make use of the following notations in the rest of the paper: $\texttt{solve}(\Sigma)$ returns $\emptyset$ if the `CNF` formula Σ is unsatisfiable, and $\texttt{solve}(\Sigma)$ returns a model of Σ otherwise. `BCP` denotes a Boolean Constraint Propagator (Zhang and Stickel 1996), which is a key component of many preprocessors. $\texttt{BCP}(\Sigma)$ 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 (Zhang and Stickel 1996; Moskewicz et al. 2001). Finally, $\Sigma[\ell \leftarrow \Phi]$ 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, and removing the valid clauses which could be generated.

Preprocessing for Model Counting

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.

For many preprocessing techniques, it can be guaranteed that the size of $p(\Sigma)$ is smaller than (or equal to) the size of Σ. A rationale for it is that the complexity of the algorithm achieving the task to be improved via preprocessing depends on the size of its input, hence 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. (Boufkhad and Roussel 2000; Liberatore 2005)). Especially, some approaches to knowledge compilation consists in adding redundant clauses to the input CNF formula in order to make it unit-refutation complete (del Val 1994; Bordeaux and Marques-Silva 2012; Bordeaux et al. 2012).

We have studied and evaluated the following elementary preprocessing techniques for model counting: vivification, occurrence reduction, backbone detection, as well as equivalence, AND, and XOR gate identification and replacement.

Vivification. Vivification (cf. Algorithm 1) (Piette, Hamadi, and Saïs 2008) 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 and the output size is always upper bounded by 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. On the one hand, if for any $j \in 1, \ldots, k$, one can prove using BCP that $\Sigma \backslash \{\alpha\} \models \ell_1 \vee \ldots \vee \ell_j$, then for sure α is entailed by $\Sigma \backslash \{\alpha\}$ so that α can be removed from Σ. On the other hand, if one can prove using BCP that $\Sigma \setminus \{\alpha\} \models \ell_1 \vee \ldots \vee \ell_j \vee \sim\ell_{j+1}$, then ℓ_{j+1} can be removed from α without questioning equivalence. Vivification is not a confluent preprocessing, i.e., both the clause ordering and the literal ordering in clauses may have an impact on the result. In our implementation, the largest clauses are handled first, and the literals are handled (line 5) based on their VSIDS (Variable State Independent, Decaying Sum) (Moskewicz et al. 2001) activities (the most active ones first).

Algorithm 1: `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
```

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 clauses by some subsuming ones. In order to determine whether a literal ℓ can be removed from a clause α of Σ, the approach consists in determining whether the clause which coincides with α except that ℓ has been replaced by $\sim\ell$ is a logical consequence of Σ. When this is the case, ℓ can be removed from α without questioning logical equivalence. Again, BCP is used as an incomplete yet efficient method to solve the entailment problem. Occurrence reduction can be viewed as a light form of vivification (since the objective is just to remove literals and not clauses). Especially, it preserves equivalence, leads to a CNF formula whose size is upper bounded by the input size and has a worst-case time complexity cubic in the input size. 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.

Algorithm 2: `occurrenceSimpl`

```
   input  : a CNF formula Σ
   output : a CNF formula equivalent to Σ
1  L← Lit(Σ);
2  while L ≠ ∅ do
3      Let ℓ ∈ L be a most frequent literal of Σ;
4      L←L \ {ℓ};
5      foreach α ∈ Σ s.t. ℓ ∈ α do
6          if ∅ ∈ BCP(Σ ∧ ℓ∧ ∼ (α \ {ℓ})) then
7              Σ←(Σ \ {α}) ∪ {α \ {ℓ}};
8  return Σ
```

Backbone identification. The backbone (Monasson et al. 1999) 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 3) is to make the backbone of the input CNF formula Σ explicit and to conjoin it to Σ. Backbone identification preserves equivalence, is space efficient (the size of the output cannot exceed the size of the input plus the number of variables of the input), but 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 (Audemard, Lagniez, and Simon 2013).

Algorithm 3: `backboneSimpl`

```
   input  : a CNF formula Σ
   output : the CNF Σ ∪ B, where B is the backbone of Σ
1  B←∅;
2  I←solve(Σ);
3  while ∃ℓ ∈ I s.t. ℓ ∉ B do
4      I'←solve(Σ∧ ∼ ℓ);
5      if I' = ∅ then B←B ∪ {ℓ}else I←I ∩ I';
6  return Σ ∪ B
```

Equivalence and gates detection and replacement. Equivalence and gates detection and replacement are preprocessing techniques which do not preserve equivalence but only the number of models of the input formula. Equivalence detection was used for preprocessing in (Bacchus and Winter 2004), while AND gates and XOR gates detection and replacement have been exploited in (Ostrowski et al. 2002).

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 of 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 (Subbarayan and Pradhan 2004), 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 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).

Equivalence detection and replacement is presented at Algorithm 4. BCP is used for detecting equivalences between literals. In the worst case, the time complexity of this preprocessing is cubic in the input size, and the output size is upper bounded by the input size.[2]

Algorithm 4: equivSimpl

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

1 $\Phi \leftarrow \Sigma$;
2 Unmark all variables of Φ;
3 **while** $\exists \ell \in Lit(\Phi)$ *s.t.* $var(\ell)$ *is not marked* **do**
4 mark $var(\ell)$;
5 $\mathcal{P}_\ell \leftarrow BCP(\Phi \wedge \ell)$;
6 $\mathcal{N}_\ell \leftarrow BCP(\Phi \wedge \sim \ell)$;
7 $\Gamma \leftarrow \{\ell \leftrightarrow \ell' | \ell' \neq \ell$ and $\ell' \in \mathcal{P}_\ell$ and $\sim \ell' \in \mathcal{N}_\ell\}$;
8 **foreach** $\ell \leftrightarrow \ell'$ **do** replace ℓ' by ℓ in Φ;
9 **return** Φ

AND gate detection and replacement is presented at Algorithm 5. In the worst case, its time complexity is cubic in the input size. At line 4, literals ℓ of $Lit(\Sigma)$ are considered w.r.t. any total ordering such that $\sim\ell$ comes just after ℓ. The test at line 7 allows for deciding whether an AND gate β with output ℓ exists in Σ. In our implementation, one tries to minimize the number of variables in this gate by taking advantage of the implication graph. The replacement of a literal ℓ by the corresponding definition β is performed (line 10) only if the number of conjuncts in the AND gate β remains "small enough" (i.e., $\leq maxA$ – in our experiments $maxA = 10$), provided that the replacement does not lead to increase the input size. This last condition ensures that the output size of the preprocessing remains upper bounded by the input size.

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

[2]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.

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 $\mathcal{P}_\ell \leftarrow (BCP(\Phi \wedge \ell) \setminus (BCP(\Phi) \cup \{\ell\})) \cup \{\sim \ell\}$;
7 **if** $\emptyset \in$ BCP$(\Phi \wedge \mathcal{P}_\ell)$ **then**
8 let $\mathcal{C}_\ell \subseteq \mathcal{P}_\ell$ s.t. $\emptyset \in$ BCP$(\Phi \wedge \mathcal{C}_\ell)$ 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

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: 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 $> maxX$). The $maxX$ condition ensures that the output size remains linear in the input size. Due to this condition, the time complexity of XORgateSimpl is in the worst case quadratic in the input size (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).

Algorithm 6: XORgateSimpl

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

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

The pmc **preprocessor.** 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. Observe that 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. Within pmc the other elementary preprocessings can be performed several times. Iteration stops when a fixpoint 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 $\|\Phi\| = \|\Sigma\|$

$\Phi \leftarrow \Sigma$;
if $optB$ **then** $\Phi \leftarrow$ backboneSimpl(Φ);
$i \leftarrow 0$;
while $i < numTries$ **do**
  $i \leftarrow i + 1$;
  if $optO$ **then** $\Phi \leftarrow$ occurrenceSimpl(Φ);
  if $optG$ **then** $\Phi \leftarrow$ gatesSimpl(Φ);
  if $optV$ **then** $\Phi \leftarrow$ vivificationSimpl(Φ);
  if *fixpoint* **then** $i \leftarrow numTries$;
return Φ

Empirical Evaluation

Setup. In our experiments, 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 and can thus be used for weighted model counting.
- $\#eq$ 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.

As to model counting, we have considered both a "direct" approach, based on the state-of-the-art model counter Cachet (www.cs.rochester.edu/~kautz/Cachet/index.htm) (Sang et al. 2004), as well as a compilation-based approach, based on the C2D compiler (reasoning.cs.ucla.edu/c2d/) which generates (smooth) d−DNNF compilations (Darwiche 2001). As explained previously, it does not make sense to use the $\#eq$ preprocessing upstream to C2D.

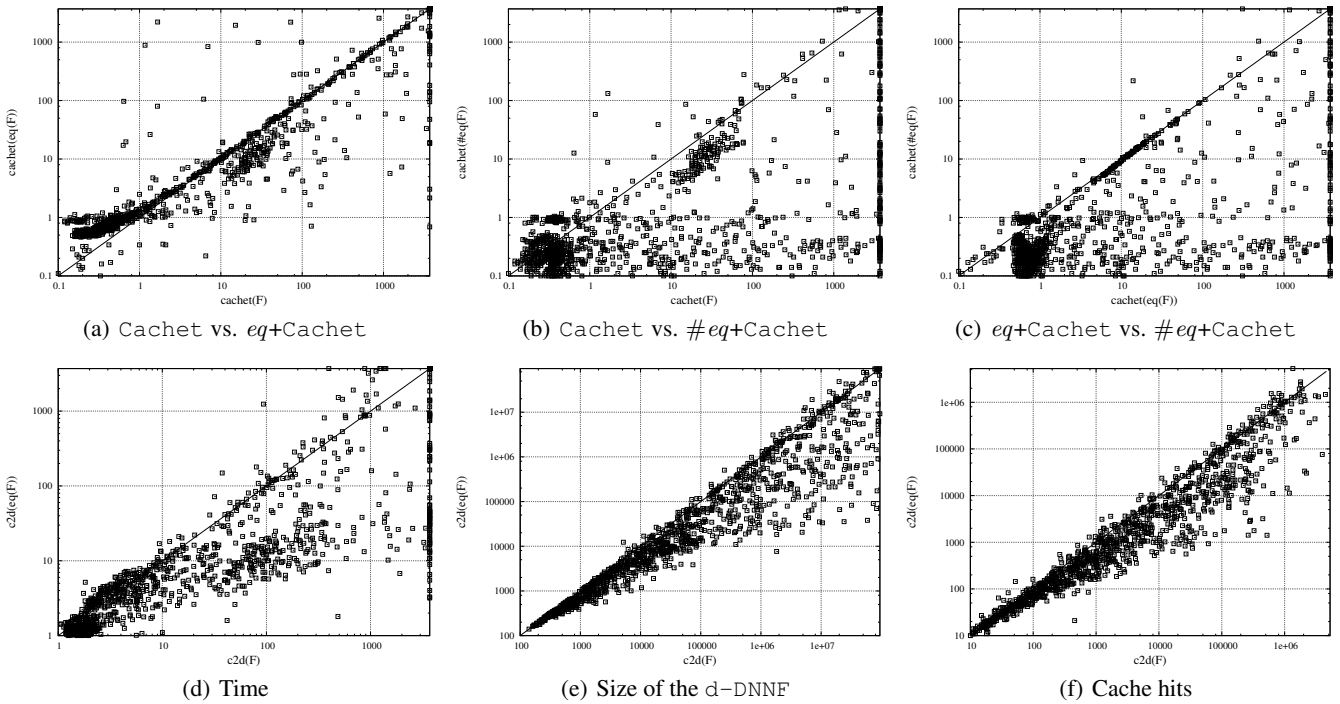
We made quite intensive experiments on a number of CNF instances Σ from different domains, available in the SAT LIBrary (www.cs.ubc.ca/~hoos/SATLIB/index-ubc.html). 1342 instances Σ from 19 families have been used. The aim was to count the number of models of each Σ using pmc for the two combinations of preprocessings listed above, and to determine whether the combination(s) under consideration prove(s) or not useful for solving it. Our experiments have been conducted on a Quad-core Intel XEON X5550 with 32GB of memory. A time-out of 3600 seconds per instance Σ has been considered for Cachet and the same time-out has been given to C2D for achieving the compilation of Σ and computing the number of models of the resulting d−DNNF formula. Our preprocessor pmc and some detailed empirical results, including the benchmarks considered in our experiments, are available at www.cril.fr/PMC/pmc.html.

Impact on Cachet and on C2D Figure 1 (a)(b)(c) illustrates the comparative performances of Cachet, being equipped (or not) with some preprocessings. No specific optimization of the preprocessing achieved depending on the family of the instance under consideration has been performed: we have considered the eq preprocessing and the $\#eq$ preprocessing for every instance. Each dot represents an instance and the time needed to solve it using the approach corresponding to the x-axis (resp. y-axis) is given by its x-coordinate (resp. y-coordinate). In part (a) of the figure, the x-axis corresponds to Cachet (without any preprocessing) while the y-axis corresponds to $\#eq$+Cachet. In part (b), the x-axis corresponds to Cachet (without any preprocessing) and the y-axis corresponds to $\#eq$+Cachet. In part (c), the x-axis corresponds to eq+Cachet and the y-axis corresponds to $\#eq$+Cachet.

In Figure 1 (d)(e)(f), the performance of C2D is compared with the one offered by eq+C2D. As on the previous figure, each dot represents an instance. Part (d) of the figure is about compilation time (plus the time needed to count the models of the compiled form), while part (e) is about the size of the resulting d−DNNF formula. Part (f) of the figure reports the number of cache hits.

Synthesis. Table 1 synthesizes some of the results. Each line compares the performances of two approaches to model counting (say, A and B), based on Cachet or C2D, and using or not some preprocessing techniques. For instance, the first line compares Cachet without any preprocessing (A) with eq+Cachet (B). #s(A or B) indicates how many instances (over 1342) 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. #s(A and B) indicates how many instances have been solved by both A and B, and T_A (resp. T_B) is the cumulated time (in seconds) used by A (resp. B) to solve them. The preprocessing time $T_{\texttt{pmc}}$ spent by pmc is included in the times reported in Figure 1 and in Table 1. $T_{\texttt{pmc}}$ is less than 1s (resp. 10s, 50s) for 80% (resp. 90%, 99%) of the instances.

The empirical results clearly show both the efficiency and the robustness of our preprocessing techniques. As to efficiency, the number of instances which can be solved within the time limit when a preprocessing is used is always higher, and sometimes significantly higher, than the the corresponding number without preprocessing. Similarly, the cumulated time needed to solve the commonly solved instances is always smaller (and sometimes significantly smaller) than



(a) Cachet vs. *eq*+Cachet (b) Cachet vs. #*eq*+Cachet (c) *eq*+Cachet vs. #*eq*+Cachet

(d) Time (e) Size of the d−DNNF (f) Cache hits

Figure 1: Comparing Cachet with (*eq* or #*eq*)+Cachet (above) and C2D with *eq*+C2D (below) on a large panel of instances from the SAT LIBrary.

A	B	#s(A or B)	#s(A) - #s(B)	#s(B) - #s(A)	#s(A and B)	T_A	T_B
Cachet	*eq*+Cachet	1047	0	28	1019	98882.6	83887.7
Cachet	#*eq*+Cachet	1151	1	132	1018	97483.9	16710.2
eq+Cachet	#*eq*+Cachet	1151	1	104	1046	111028.0	18355.4
C2D	*eq*+C2D	1274	7	77	1190	123923.0	53653.2

Table 1: A synthesis of the empirical results about the impact of preprocessing on Cachet and C2D

the corresponding time without any preprocessing. Interestingly, *eq*+C2D also leads to substantial space savings compared to C2D (our experiments showed that the size of the resulting d−DNNF formulae can be more than one order of magnitude larger without preprocessing, and that the cumulated size is more than 1.5 larger). This is a strong piece of evidence that the practical impact of pmc is not limited to the model counting issue, and that the *eq* preprocessing can also prove useful for equivalence-preserving knowledge compilation. As to robustness, the number of instances solved within the time limit when no preprocessing has been used and not solved within it when a preprocessing technique is considered remains very low, whatever the approach to model counting under consideration. Finally, one can also observe that the impact of the equivalence/gates detection and replacement is huge (#*eq*+Cachet is a much better performer than *eq*+Cachet).

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.

This work opens several perspectives for further research. Beyond size reduction, it would be interesting to determine some explanations to the improvements achieved by taking advantage of pmc (for instance, whether they can lead to a significant decrease of the treewidth of the input CNF formula). It would be useful to determine the "best" combinations of elementary preprocessings, depending on the benchmark families. It would be nice to evaluate the impact of pmc when other approaches to model counting are considered, especially approximate model counters (Wei and Selman 2005; Gomes and Sellmann 2004) and other compilation-based approaches (Koriche et al. 2013; Bryant 1986; Darwiche 2011). Assessing whether pmc prove useful upstream to other knowledge compilation techniques (for instance (Boufkhad et al. 1997; Subbarayan, Bordeaux, and Hamadi 2007; Fargier and Marquis 2008; Bordeaux and Marques-Silva 2012)) would also be valuable.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments. This work is partially supported by the project BR4CP ANR-11-BS02-008 of the French National Agency for Research.

References

Apsel, U., and Brafman, R. I. 2012. Lifted MEU by weighted model counting. In *Proc. of AAAI'12*.

Audemard, G., and Simon, L. 2009. Predicting learnt clauses quality in modern sat solver. In *Proc. of IJCAI'09*, 399–404.

Audemard, G.; Lagniez, J.-M.; and Simon, L. 2013. Just-in-time compilation of knowledge bases. In *Proc. of IJCAI'13*, 447–453.

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

Bordeaux, L., and Marques-Silva, J. 2012. Knowledge compilation with empowerment. In *Proc. of SOFSEM'12*, 612–624.

Bordeaux, L.; Janota, M.; Marques-Silva, J.; and Marquis, P. 2012. On unit-refutation complete formulae with existentially quantified variables. In *Proc. of KR'12*, 75–84.

Boufkhad, Y., and Roussel, O. 2000. Redundancy in random SAT formulas. In *Proc. of AAAI'00*, 273–278.

Boufkhad, Y.; Grégoire, E.; Marquis, P.; Mazure, B.; and Saïs, L. 1997. Tractable cover compilations. In *Proc. of IJCAI'97*, 122–127.

Bryant, R. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers* C-35(8):677–692.

Chavira, M., and Darwiche, A. 2008. On probabilistic inference by weighted model counting. *Artificial Intelligence* 172(6-7):772–799.

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

Darwiche, A. 2011. SDD: A new canonical representation of propositional knowledge bases. In *Proc. of IJCAI'11*, 819–826.

del Val, A. 1994. Tractable databases: How to make propositional unit resolution complete through compilation. In *Proc. of KR'94*, 551–561.

Domshlak, C., and Hoffmann, J. 2006. Fast probabilistic planning through weighted model counting. In *Proc. of ICAPS'06*, 243–252.

Een, N., and Biere, A. 2005. Effective preprocessing in sat through variable and clause elimination. In *Proc. of SAT'05*, 61–75.

Fargier, H., and Marquis, P. 2008. Extending the knowledge compilation map: Krom, Horn, affine and beyond. In *Proc. of AAAI'08*, 442–447.

Gomes, C., and Sellmann, M. 2004. Streamlined constraint reasoning. In *Proc. of CP'04*. 274–289.

Han, H., and Somenzi, F. 2007. Alembic: An efficient algorithm for cnf preprocessing. In *Proc. of DAC'07*, 582–587.

Heule, M.; Järvisalo, M.; and Biere, A. 2010. Clause elimination procedures for cnf formulas. In *Proc. of LPAR'10*, 357–371.

Heule, M. J. H.; Järvisalo, M.; and Biere, A. 2011. Efficient cnf simplification based on binary implication graphs. In *Proc. of SAT'11*, 201–215.

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

Koriche, F.; Lagniez, J.-M.; Marquis, P.; and Thomas, S. 2013. Knowledge compilation for model counting: Affine decision trees. In *Proc. of IJCAI'13*, 947–953.

Liberatore, P. 2005. Redundancy in logic I: CNF propositional formulae. *Artificial Intelligence* 163(2):203–232.

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

Monasson, R.; Zecchina, R.; Kirkpatrick, S.; Selman, B.; and Troyansky, L. 1999. Determining computational complexity from characteristic 'phase transitions'. *Nature* 33:133–137.

Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an efficient sat solver. In *Proc. of DAC '01*, 530–535.

Ostrowski, R.; Grégoire, E.; Mazure, B.; and Saïs, L. 2002. Recovering and exploiting structural knowledge from cnf formulas. In *Proc. of CP'02*, 185–199.

Palacios, H.; Bonet, B.; Darwiche, A.; and Geffner, H. 2005. Pruning conformant plans by counting models on compiled d-DNNF representations. In *Proc. of ICAPS'05*, 141–150.

Piette, C.; Hamadi, Y.; and Saïs, L. 2008. Vivifying propositional clausal formulae. In *Proc. of ECAI'08*, 525–529.

Roth, D. 1996. On the hardness of approximate reasoning. *Artificial Intelligence* 82(1–2):273–302.

Sang, T.; Beame, P.; and Kautz, H. A. 2005. Performing Bayesian inference by weighted model counting. In *Proc. of AAAI'05*, 475–482.

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

Subbarayan, S., and Pradhan, D. K. 2004. Niver: Non increasing variable elimination resolution for preprocessing sat instances. In *Proc. of SAT'04*, 276–291.

Subbarayan, S.; Bordeaux, L.; and Hamadi, Y. 2007. Knowledge compilation properties of tree-of-BDDs. In *Proc. of AAAI'07*, 502–507.

Valiant, L. G. 1979. The complexity of computing the permanent. *Theoretical Computer Science* 8:189–201.

Wei, W., and Selman, B. 2005. A new approach to model counting. In *Proc. of SAT'05*, 324–339.

Zhang, H., and Stickel, M. E. 1996. An efficient algorithm for unit propagation. In *Proc. of ISAIM96*, 166–169.