A Recursive Algorithm for Projected Model Counting

Jean-Marie Lagniez and **Pierre Marquis**[1]
CRIL, U. Artois & CNRS
Institut Universitaire de France[1]
F-62300 Lens, France
{lagniez, marquis}@cril.fr

Abstract

We present a recursive algorithm for projected model counting, i.e., the problem consisting in determining the number of models $\|\exists X.\Sigma\|$ of a propositional formula Σ after eliminating from it a given set X of variables. Based on a "standard" model counter, our algorithm projMC takes advantage of a disjunctive decomposition scheme of $\exists X.\Sigma$ for computing $\|\exists X.\Sigma\|$. It also looks for disjoint components in its input for improving the computation. Our experiments show that in many cases projMC is significantly more efficient than the previous algorithms for projected model counting from the literature.

Introduction

In this paper, we are concerned with the projected model counting problem. Given a propositional formula Σ and a set X of propositional variables to be forgotten, one wants to compute the number of interpretations of the variables occurring in Σ but not in X, which coincide on X with a model of Σ. Stated otherwise, the objective is to count the number of models of the quantified Boolean formula $\exists X.\Sigma$ over its variables (i.e., the variables occurring in Σ but not in X).

The projected model counting problem is a central issue to a number of AI problems (for instance, in planning, when the objective is to compute the robustness of a given plan given by the number of initial states from which the execution of the plan reaches a goal state (Aziz et al. 2015)), but also outside AI (especially it proves useful in some formal verification problems (Klebanov, Manthey, and Muise 2013)).

Since it generalizes the standard model counting problem (recovered when $X = \emptyset$), the projected model counting problem is at least as hard as the latter (#P-hard). The presence of variables X to be forgotten nevertheless may render the problem easier in some cases (thus, when every variable of Σ belongs to X, the problem boils down to deciding the satisfiability of Σ). That said, a naive approach which would consist first in forgetting the variables of X from Σ, then in counting the number of models of the resulting formula would be impractical in many cases, in particular when X is large. Indeed, forgetting the variables of X from Σ in a brute-force way often leads to a formula which is much larger than Σ (in the worst case, an exponential blow-up may occur).

Despite the importance of the problem, few algorithms for projected model counting have been pointed out so far. An FPT algorithm, where the parameter is the treewidth of the primal graph of the input instance, has been designed recently (Fichte et al. 2018), but this algorithm is practical only for instances having a small treewidth. Three other algorithms for the projected model counting task have been presented in (Aziz et al. 2015), namely dSharp$_{\mathsf{P}}$, #clasp, and d2c. Those algorithms are quite dissimilar in essence:

- dSharp$_{\mathsf{P}}$ is an adaptation of the model counter dSharp (Muise et al. 2012), which computes a Decision-DNNF representation of its input Σ to determine the number of models $\|\Sigma\|$. dSharp$_{\mathsf{P}}$ considers as an additional input a set P of protected variables (i.e., those variables of Σ which should not be forgotten). The search achieved by dSharp is constrained so that the decision variables are taken in priority from P. Whenever there is no variable of P in the current formula (i.e., this formula contains only variables from X), a sat solver is used to determine whether it is consistent. If so, its number of models is 1, otherwise it is equal to 0. This technique has also been considered in (Klebanov, Manthey, and Muise 2013). Note that the constraint imposed on the variable ordering limits the ability to find cutsets of small size, and this may have a strong (yet negative) impact on the quality of the conjunctive decompositions of Σ found by dSharp.

- #clasp is an extension of clasp, an algorithm for model enumeration on a projected set of variables (Gebser, Kaufmann, and Schaub 2009), which basically amounts to computing a deterministic DNF representation of $\exists X.\Sigma$. Each time an implicant of $\exists X.\Sigma$ is found, a blocking clause equivalent to the negation of this implicant is added to Σ. In #clasp, whenever an implicant of $\exists X.\Sigma$ is found, a prime implicant is first extracted from it in a greedy fashion (this is reminiscent to the approach considered in (Schrag 1996; Castell 1996)). Compared to clasp, this further extraction step often leads to the adding of far less blocking clauses to Σ, so that #clasp performs usually better than clasp.

- d2c consists in computing first a Decision-DNNF representation of Σ, then forgetting in it all the variables of X (this can be achieved in linear time, but the resulting representation is not deterministic any longer in the general case). The next step consists in turning this resulting representation into a CNF one. This can be done in linear time via the introduction of new variables while preserving the number of models of the input (Tseitin 1968). Finally, the number of models of the resulting CNF formula is evaluated using sharpSAT (Thurley 2006).

Unlike those algorithms, our algorithm for projected model counting, called projMC, is a recursive algorithm exploiting a disjunctive decomposition scheme of $\exists X.\Sigma$ for computing $\|\exists X.\Sigma\|$. More precisely, $\exists X.\Sigma$ is split into an equivalent (disjunctively interpreted) set of pairwise inconsistent formulae, so that $\|\exists X.\Sigma\|$ can be computed by summing up the corresponding projected model counts. projMC also looks for disjoint components in its input for improving the computation.

To evaluate the performance of our approach, we measured the time required by projMC for achieving the projected model counting task on a number of benchmarks (uniform random 3-CNF formulae, random Boolean circuits, planning instances). Our experiments show that in many cases projMC challenges the previous algorithms for projected model counting from the literature. For some benchmarks, when compared to dSharp$_\mathsf{P}$, #clasp, and d2c, the time savings achieved by projMC are of several orders of magnitude. In order to verify that the improvements obtained in practice with projMC are actually due to the underlying approach and not to the performance of the "standard" model counter used in it (which is D4 (Lagniez and Marquis 2017) in our implementation), we also developed D4$_\mathsf{P}$, which is the same algorithm as dSharp$_\mathsf{P}$, but using D4 as a model counter instead of dSharp. While D4$_\mathsf{P}$ performs typically better than dSharp$_\mathsf{P}$ (and #clasp and d2c), projMC turns out to be a better performer than D4$_\mathsf{P}$ for many instances.

The rest of the paper is organized as follows. In the next section, we present some formal preliminaries. Then we describe our new projected model counter. Afterwards we present the empirical protocol which has been considered in the experiments, as well as the corresponding experimental results. Finally, a last section concludes the paper and gives some perspectives for further research. The binary code of projMC, as well as the benchmarks used in our experiments and additional empirical results are available from www.cril.fr/KC/.

Formal Preliminaries

Let $\mathcal{L}_\mathcal{P}$ be a propositional language built up from a finite set of propositional variables $\mathcal{P}$ and the usual connectives. $\bot$ (resp. $\top$) is the Boolean constant always false (resp. true). An *interpretation* (or world) ω is a mapping from $\mathcal{P}$ to $\{0,1\}$.The set of all interpretations is denoted $\mathcal{W}$. An interpretation ω is a *model* of a formula $\varphi \in \mathcal{L}_\mathcal{P}$ if and only if it makes it true in the usual truth functional way. $Mod(\varphi)$ denotes the set of models of the formula φ, i.e., $Mod(\varphi) = \{\omega \in \mathcal{W} \mid \omega \models \varphi\}$. $\models$ denotes logical entailment and $\equiv$ logical equivalence, i.e., $\varphi \models \psi$ iff $Mod(\varphi) \subseteq Mod(\psi)$ and $\varphi \equiv \psi$ iff $Mod(\varphi) = Mod(\psi)$. When $\varphi \in \mathcal{L}_\mathcal{P}$ and $X \subseteq \mathcal{P}$, $\exists X.\varphi$ is a quantified Boolean formula denoting (up to logical equivalence) the most general consequence of φ which is independent from the variables of X (see e.g., (Lang, Liberatore, and Marquis 2003)). $Var(\varphi)$ denotes the set of variables occurring in $\varphi \in \mathcal{L}_\mathcal{P}$ and $Var(\{\varphi_1, \ldots, \varphi_k\}) = \bigcup_{i=1}^{k} Var(\varphi_i)$; when $X \subseteq \mathcal{P}$, we have $Var(\exists X.\varphi) = Var(\varphi) \setminus X$.

A *literal* ℓ is a propositional variable or a negated one. When ℓ is a literal over x, its *complementary literal* $\sim\ell$ is given by $\sim\ell = \neg x$ if $\ell = x$ and $\sim\ell = x$ if $\ell = \neg x$, and we note $var(\ell) = x$. A literal ℓ is *pure* in a CNF formula Σ when Σ contains no occurrence of $\sim\ell$. One also says that the corresponding variable $var(\ell)$ is pure in Σ. A *term* is a conjunction of literals. It is also viewed as the set of its literals when this is convenient. A *clause* is a disjunction of literals. When $\delta = \bigvee_{j=1}^{m} \ell_j$ is a clause, $\sim\delta$ denotes the term given by $\bigwedge_{j=1}^{m} \sim\ell_j$. A clause δ_1 is a *subclause* of a clause δ_2 when every literal of δ_1 is a literal of δ_2. A CNF formula φ is a conjunction of clauses. It is also viewed as the set of its clauses when this is convenient. $\|\varphi\|$ denotes the number of models of φ over $Var(\varphi)$.

Given a subset X of $\mathcal{P}$ and a world ω, $\omega[X]$ is the term $\bigwedge_{x \in X | \omega \models x} x \wedge \bigwedge_{x \in X | \omega \models \neg x} \neg x$. The *conditioning* of a CNF formula φ by a consistent term γ is the CNF formula $\varphi \mid \gamma$ obtained from φ by removing each clause containing a literal of γ and by shortening the remaining clauses, removing from them the complementary literals of those of γ.

BCP denotes a *Boolean Constraint Propagator* (Zhang and Stickel 1996; Moskewicz et al. 2001), which is a key component of many solvers. BCP(Σ) returns the CNF formula Σ once simplified using unit propagation.

The *primal graph* of a CNF formula Σ is the (undirected) graph where vertices correspond to the variables of Σ and an edge connecting two variables exists whenever one can find a clause of Σ where both variables occur. Every connected component of this graph (i.e., a maximal subset of vertices which are pairwise connected by a path) corresponds to a subset of clauses of Σ, referred to as a *connected component* of the formula Σ.

A Recursive Algorithm for Projected Model Counting

Our projected model counter projMC computes $\|\exists X.\Sigma\|$ where Σ is a CNF formula and X a set of propositional variables. Assuming Σ being in CNF is harmless since one can use Tseitin technique (Tseitin 1968) to associate in linear time with any propositional circuit into a CNF formula having the same number of models.

Unlike the previous algorithms for projected model counting sketched in the introductive section, projMC is a recursive algorithm guided by a *deterministic disjunctive form* for Σ w.r.t. X. Such a form is a (disjunctively interpreted) set of formulae $\{\varphi_1, \ldots, \varphi_{k+1}\}$ over $Var(\Sigma)$ satisfying the following conditions:

- $\forall i \in \{1,\ldots,k+1\}$, $Var(\varphi_i) \cap X = \emptyset$;
- $\forall i,j \in \{1,\ldots,k+1\}$, if $i \neq j$ then $\varphi_i \wedge \varphi_j \models \bot$;
- $\bigvee_{i=1}^{k+1} \varphi_i \equiv \top$.

Because $\bigvee_{i=1}^{k+1} \varphi_i$ is valid, we have $\exists X.\Sigma \equiv (\exists X.\Sigma) \wedge (\bigvee_{i=1}^{k+1} \varphi_i) \equiv \bigvee_{i=1}^{k+1} (\exists X.\Sigma) \wedge \varphi_i \equiv \bigvee_{i=1}^{k+1} \exists X.(\Sigma \wedge \varphi_i)$ since no element of a deterministic disjunctive form for Σ w.r.t. X contains a variable of X. Furthermore, since the elements of a deterministic disjunctive form for Σ w.r.t. X are pairwise conflicting, we get that $\|\exists X.\Sigma\| = \sum_{i=1}^{k+1} \|(\exists X.\Sigma) \wedge \varphi_i\|$, hence

$$\|\exists X.\Sigma\| = \sum_{i=1}^{k+1} \|\exists X.(\Sigma \wedge \varphi_i)\|.$$

The set $\{\Sigma \wedge \varphi_i \mid i \in \{1,\ldots,k+1\}\}$ is referred to as *the disjunctive decomposition* associated with $\{\varphi_1,\ldots,\varphi_{k+1}\}$.

Technically speaking, such a notion of disjunctive decomposition can be related to several concepts considered in some previous works about SAT or #SAT. On the one hand, it can be viewed as a specific case of the notion of set of scattered formulae from a formula Σ (see (Hyvärinen, Junttila, and Niemelä 2006) for details), obtained by focusing on clauses/terms not containing any variable from X. However, the objective pursued in (Hyvärinen, Junttila, and Niemelä 2006) was quite distinct from our own one (in this paper, the decomposition was used as a distribution method for SAT solving in grids). On the other hand, when it consists of consistent formulae, a deterministic disjunctive form for Σ w.r.t. X forms a partition, just as the primes of an X-partition of Σ, used for generating an SDD representation of Σ (see (Darwiche 2011) for details). Nevertheless, the connection to SDD does not extend further: whenever Σ contains variables from X and from $Var(\Sigma) \setminus X$, the disjunctive decomposition associated with a deterministic disjunctive form is not a $(X, Var(\Sigma) \setminus X)$-decomposition of Σ: disjunctive decompositions do not aim to create formulae that do not share variables.

Clearly enough, generating a disjunctive decomposition $\{\Sigma \wedge \varphi_i \mid i \in \{1,\ldots,k+1\}\}$ associated with a deterministic disjunctive form for Σ w.r.t. X is computationally useful for computing $\|\exists X.\Sigma\|$ only if the formulae $\Sigma \wedge \varphi_i$ are somewhat more simple than Σ. To ensure it and guarantee the termination of projMC, one does not compute any deterministic disjunctive form for Σ w.r.t. X but one *induced by a model* ω of Σ (if there is not such model, then $\exists X.\Sigma$ is inconsistent and $\|\exists X.\Sigma\| = 0$). One starts by considering the CNF formula $\Sigma \mid \omega[X] = \bigwedge_{i=1}^{k} \delta_i$, called the *core* φ_1 of the deterministic disjunctive form $\{\varphi_1,\ldots,\varphi_{k+1}\}$ for Σ w.r.t. X induced by ω. By construction, $\varphi_1 = \bigwedge_{i=1}^{k} \delta_i$ contains variables occurring in $Var(\Sigma)$ but not in X. Then we generate a (disjunctively interpreted) set of CNF formulae $\{\varphi_2,\ldots,\varphi_{k+1}\}$ which is equivalent to the negation of φ_1, by defining, for each $i \in \{1,\ldots,k\}$, $\varphi_{i+1} = (\bigwedge_{j=1}^{i-1} \delta_j) \wedge \sim\delta_i$. By construction, $\{\varphi_1,\ldots,\varphi_{k+1}\}$ satisfies the conditions of a deterministic disjunctive form for Σ w.r.t. X. Note also that $\exists X.(\Sigma \wedge \varphi_1)$ is equivalent to φ_1. Indeed, $\exists X.(\Sigma \wedge \varphi_1)$ is equivalent to $(\exists X.\Sigma) \wedge \varphi_1$ since φ_1 does not contain any variable of X. Furthermore, since $\omega[X] \wedge \Sigma \models \Sigma$, we have that $\exists X.(\omega[X] \wedge \Sigma) \models \exists X.\Sigma$. But $\exists X.(\omega[X] \wedge \Sigma)$ is equivalent to $\Sigma \mid \omega[X]$, hence to φ_1, so $\varphi_1 \models \exists X.\Sigma$, and as a consequence $(\exists X.\Sigma) \wedge \varphi_1$ is equivalent to φ_1. Accordingly, when computing the number of models of the disjunctive decomposition $\{\Sigma \wedge \varphi_i \mid i \in \{1,\ldots,k+1\}\}$ associated with $\{\varphi_1,\ldots,\varphi_{k+1}\}$, one replaces $\Sigma \wedge \varphi_1$ by φ_1. Since $Var(\varphi_1) \cap X = \emptyset$, one can take advantage of a "standard" model counter for computing $\|\varphi_1\|$ and no recursive call of projMC is needed (this is a base case for the recursion).

Our algorithm projMC also takes advantage of any conjunctive decomposition of its input Σ into disjoint connected components whenever such a decomposition exists. Indeed, whenever Σ can be split into two sets of clauses α and β not sharing any variable so that $\Sigma \equiv \alpha \wedge \beta$, then the computation of $\exists X.\Sigma$ can be achieved by computing $\exists X.\alpha$ and $\exists X.\beta$ since $\exists X.\Sigma \equiv \exists X.(\alpha \wedge \beta) \equiv (\exists X.\alpha) \wedge (\exists X.\beta)$. And since $\exists X.\alpha$ and $\exists X.\beta$ do not share any variable, one can compute $\|\exists X.\Sigma\| = \|\exists X.\alpha\| \times \|\exists X.\beta\|$. As it is the case for the model counting problem, taking advantage of conjunctive decompositions proves to be very useful for the efficiency purpose.

Algorithm 1: projMC(Σ, X)

```
   input : a CNF formula Σ
   input : a set of variables X to be forgotten
   output: the number of models of ∃X.Σ over
           Var(Σ) \ X
1  Σ ← BCP(Σ)
2  if Var(Σ) ∩ X = ∅ then return MC(Σ)
3  if cache(Σ) ≠ nil then return cache(Σ)
4  comps ← connectedComponents(Σ)
5  if #(comps) ≠ 1 then
6  |  cpt ← 1
7  |  foreach φ ∈ comps do
8  |  |  cpt ← cpt× projMC(φ, X)
9  else
10 |  ω ← sat(Σ)
11 |  cpt ← 0
12 |  if ω = ∅ then break
13 |  dd ← DD(Σ, ω, X)
14 |  foreach φ ∈ dd do
15 |  |  cpt ← cpt+
   |  |        projMC(φ, X) ×2^#(Var(dd)\(Var(φ)∪X))
16 cache(Σ) ← cpt
17 return cpt
```

More in detail, Algorithm 1 provides the pseudo-code of the projected model counter projMC. The projected model counting of a given CNF formula Σ w.r.t. a set of variables X to be forgotten is achieved by calling projMC(Σ, X). At line 1, Σ is simplified using Boolean constraint propagation. At line 2, we consider the case when Σ does not contain any variable of X. In this case, there is no variable to be forgot-

ten and it is enough to compute the number of models of Σ, using any model counter MC. This is a base case of our algorithm (no recursion takes place). At line 3 one looks into a cache to check whether or not the current formula Σ has already been encountered during the search. The cache, initially empty, gathers pairs consisting of a CNF formula and the corresponding projected model count w.r.t. X. Each time Σ has already been cached, instead of re-computing $\|\exists X.\Sigma\|$ from scratch, it is enough to return $\mathsf{cache}(\Sigma)$. In our implementation, the cache is residual and shared with the model counter MC. At line 4, one partitions Σ into a set of CNF formulae that are pairwise variable-independent (i.e., two distinct elements of *comps* are built up from two disjoint sets of variables). connectedComponents is a "standard" procedure used in previous model counters. It is based on the search for the connected components of the primal graph of Σ and it returns a set *comps* of CNF formulae composed of clauses from Σ, so that every pair of distinct formulae from *comps* do not share any common variable. At line 5, one tests how many connected components have been found in Σ. If there are more than one component, then at lines 6, 7, and 8, one recursively computes the projected model counts corresponding to each of them, and we store in cpt the product of the corresponding counts. For efficiency reasons, in our implementation, one uses a specific trick that it is not made explicit in the algorithm for the sake of readibility: systematically, one sets aside all the components forming a consistent term containing as many literals as possible. For instance, if the input Σ is equal to $x_1 \wedge \neg x_2 \wedge (x_3 \vee x_4)$, one sets aside the two components x_1 and $\neg x_2$ to keep only the component $x_3 \vee x_4$. Indeed, for a set of components forming a consistent term like $\{x_1, \neg x_2\}$, the corresponding projected model count is always equal to 1 (whatever the variables occurring in the term belongs to X or not). Hence projMC returns directly 1 in this case, without needing to consider the literals of the term independently in distinct recursive calls. At line 9, one considers the remaining case when only one component has been found (i.e., no decomposition exists). At line 10, one looks for a model ω of Σ over $Var(\Sigma)$. If no such model exists ($\omega = \emptyset$), then Σ is inconsistent, hence so is $\exists X.\Sigma$, and the algorithm exits from the conditional (line 12). The heuristic used by the solver sat tries to satisfy Σ by assigning in priority the variables from X. An interesting consequence of this heuristic is that it leads to a lazy handling of the clauses of Σ which contain a literal ℓ pure in Σ and such that $var(\ell) = x \in X$. Indeed, when ℓ is pure in Σ, ℓ will be set to 1 by ω so that no clauses of Σ containing ℓ will belong to $\Sigma \mid \omega[X]$. This does not invalidate the correctness of the approach since such clauses can be removed from Σ without changing its projected model count. Thus, while the "standard" pure literal rule (i.e., the one consisting in removing every clause of Σ containing a variable which is pure in Σ) cannot be applied safely (it does not preserve the number of models of its input in the general case), its restriction where only pure literals over X are considered is correct: let α (resp. β) be the subset of the clauses of Σ containing ℓ (resp. not containing ℓ). We have $\exists X.\Sigma \equiv \exists X.(\exists\{x\}.(\alpha \wedge \beta))$. Since ℓ is pure in Σ, x does not occur in β. Hence $\exists\{x\}.(\alpha \wedge \beta) \equiv (\exists\{x\}.\alpha) \wedge \beta$. Now, since every clause of α contains ℓ, $\exists\{x\}.\alpha$ is valid, therefore $\exists X.\Sigma \equiv \exists X.\beta$. At line 13, once computes the disjunctive decomposition dd associated with the deterministic disjunctive form for Σ w.r.t. X induced by ω. At lines 11, 14, and 15, one sums the projected model counts for the formulae belonging to dd (note that when the elements φ of dd do not contain the same set of variables, a preliminary normalization step – multiplying each count by 2 to the power of the number of variables occurring in dd but not in φ and not in X – before summing up, is necessary). Finally, at line 16, one adds Σ to the cache, associated with the corresponding projected model count cpt, and at line 17, one returns the value of cpt.

Algorithm 2: $\mathsf{DD}(\Sigma, \omega, X)$

input : a CNF formula Σ
input : a model ω of Σ over $Var(\Sigma)$
input : a set of variables X to be forgotten
output: a set dd of CNF formulae, the disjunctive decomposition associated with the deterministic disjunctive form for Σ w.r.t. X induced by ω

1 $\{\delta_1, \ldots, \delta_k\} \leftarrow \mathsf{BCP}(\Sigma \mid \omega[X])$
2 $dd \leftarrow \{\bigwedge_{i=1}^{k} \delta_i\}$
3 foreach $\delta_j \in \{\delta_1, \ldots, \delta_k\}$ do
4 $\quad dd \leftarrow dd \cup \{\Sigma \wedge (\bigwedge_{l=1}^{j-1} \delta_l) \wedge \sim\delta_j\}$
5 return dd

Algorithm 2 provides the pseudo-code of the program which generates the disjunctive decomposition associated with the deterministic disjunctive form of Σ w.r.t. X induced by the model ω of Σ used for guiding the search. At line 1, Σ is first conditioned by the consistent term $\omega[X]$. This means that every clause of Σ containing a literal belonging to $\omega[X]$ is removed from Σ, and in the remaining clauses, every literal ℓ over X such that $\sim\ell$ is a literal of $\omega[X]$ is removed. Finally, Boolean constraint propagation is applied to $\Sigma \mid \omega[X]$ in order to simplify it further (if possible). Clearly, no variable of X occurs in the resulting (conjunctively interpreted) set of clauses $\{\delta_1, \ldots, \delta_k\}$, which is the core of the deterministic disjunctive form for Σ w.r.t. X induced by ω that is computed. At line 2, we first initialize the disjunctive decomposition dd as the singleton containing the core. At lines 3 and 4, we add to dd k CNF formulae since the core contains k clauses. At line 4, the clauses of Σ subsumed by the clauses δ_l are removed (this is not detailed in the pseudo-code for the sake of readability). By construction, the formulae of dd are pairwise inconsistent. Furthermore, every δ_j of the core is a subclause of a clause of Σ since the core is obtained by conditioning Σ using a consistent term. Accordingly, in our implementation, when δ_j is equal to a clause of Σ, δ_j is not taken into consideration within the loop (indeed, in this case, $\Sigma \wedge \delta_j$ is equivalent to Σ and $\Sigma \wedge \sim\delta_j$ is inconsistent, so that its contribution to the total model count will be equal to 0). Finally, at line 5, one returns the disjunctive decomposition dd which has been computed.

An Example. Here is an example illustrating how projMC works. Let $\Sigma = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_3 \vee x_5)$. Let $X = \{x_2, x_3\}$. The formula $\exists X.\Sigma$ is equivalent to $x_1 \vee x_4 \vee x_5$, so that $\|\exists X.\Sigma\| = 7$.

When run on Σ and X, the first instructions of projMC have no effect (applying BCP does not change Σ, and Σ has a unique connected component). Suppose that the model ω of Σ setting every variable to 1 has been found at line 10. Then the core $\Sigma \mid \omega[X]$ of the deterministic disjunctive form for Σ w.r.t. X induced by ω will be equal to x_5. As a consequence, the other formula belonging to the disjunctive decomposition associated with this deterministic disjunctive form will be $\Sigma \wedge \neg x_5$.

The recursive call of projMC on the core x_5 and X will lead to calling MC on x_5 as expected (since no variable of X occurs in the core). Then MC returns 1, and finally the recursive call of projMC on x_5 returns $1 \times 2^2 = 4$ since the two variables x_1, x_4 belong to the set of variables of dd but not to the set of variables of the associated core (this normalization step is achieved at line 15).

The recursive call of projMC on $\Sigma \wedge \neg x_5$ and X leads first to simplify $\Sigma \wedge \neg x_5$ using BCP, so that the formula $\neg x_5 \wedge (x_1 \vee x_2) \wedge (\neg x_2 \vee x_4) \wedge \neg x_3$ is got. Since the two components $\neg x_5$ and $\neg x_3$ form together a consistent term, only one component actually needs to be considered, namely $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_4)$.

So, at that step, projMC is called on $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_4)$ and X. Suppose again that the model ω of Σ setting every variable to 1 has been found. Then the resulting core will be equal to x_4, and the resulting disjunctive decomposition will be $\{x_4, (x_1 \vee x_2) \wedge (\neg x_2 \vee x_4) \wedge \neg x_4\}$. Since $\neg x_4$ does not contain a variable of X, at the next recursive call to projMC, MC is used to compute its model counts, equal to 1, and finally the recursive call of projMC on x_4 returns $1 \times 2^1 = 2$ since the variable x_1 belongs to the set of variables of the disjunctive decomposition but not to the set of variables of the associated core. The recursive call of projMC on $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_4) \wedge \neg x_4$ and X leads first to simplify (using BCP) the input into $x_1 \wedge \neg x_2 \wedge \neg x_4$, which is a consistent term. Hence it has only 1 model. Thus one obtains that $\|\exists X.((x_1 \vee x_2) \wedge (\neg x_2 \vee x_4))\| = 2 + 1 = 3$. Finally the previously computed counts are summed up. One thus gets $4 + 3 = 7$ models, as expected.

Interestingly, it can be observed that projMC handles adequately the cases when $X = \emptyset$ or $X = Var(\Sigma)$, in the sense that it does not waste too much time in many recursive calls for any of those two "extreme" situations. Indeed, when $X = \emptyset$, projMC mainly boils down to calling a "standard" model counter MC (line 2), as expected. When $X = Var(\Sigma)$, if Σ is unsatisfiable, then it will be detected as such during the first call to projMC (line 12) and a count of 0 will be returned. In the remaining case, a model ω of Σ will be found. Because $\omega[X]$ coincides with ω when $X = Var(\Sigma)$, the core of the deterministic disjunctive form for Σ w.r.t. X induced by ω will be equal to the empty set of clauses ($\Sigma \mid \omega$ is valid when ω is a model of Σ), so that the disjunctive decomposition associated with this deterministic disjunctive form will consist only of this core. Since this core contains no variable, the next recursive call to projMC (i.e., the one with the core as an operand) will mainly consist in calling the model counter MC (line 2) on it, and MC will return 1 in this case.

Finally, one can prove that our algorithm projMC actually does the job for which it has been designed:

Proposition 1 *Algorithm 1 is correct and terminates.*

Proof: The correctness of projMC (i.e., the fact that the result provided is equal to $\|\exists X.\Sigma\|$ on inputs Σ and X) comes from the fact that each rule used in the algorithm is sound w.r.t. the projected model counting task, as explained previously. The key equalities are $\|\exists X.\Sigma\| = \sum_{i=1}^{k+1} \|\exists X.(\Sigma \wedge \varphi_i)\|$ for any deterministic disjunctive form $\{\varphi_1, \ldots, \varphi_{k+1}\}$ for Σ w.r.t. X, and $\|\exists X.(\alpha \wedge \beta)\| = \|\exists X.\alpha\| \times \|\exists X.\beta\|$ when $Var(\alpha) \cap Var(\beta) = \emptyset$.

The termination of projMC comes from the fact that each time a deterministic disjunctive form for Σ w.r.t. X is computed, its core does not contain any variable of X, so that the recursive call to projMC concerning it will lead to the base case of the recursion (line 2). As to the other disjoints $\Sigma \wedge \varphi_i$ of the corresponding disjunctive decomposition, by construction, every φ_i contains the negation of a subclause δ_{i-1} of Σ. Hence, at the next call to projMC concerning this disjoint, the literals of the term $\sim\delta_{i-1}$ will be assigned and the corresponding variables will be removed from the input using Boolean constraint propagation (line 1). Thus the number of variables of the input strictly diminishes at each step and this ensures the termination of the algorithm. ■

Note that the detection of disjoint components (lines 4 to 9 in Algorithm 1) coud be frozen in projMC without questioning the correctness and the termination of this algorithm (however, it has a significant impact on the efficiency of the computation on some instances). Similarly, the use of a cache has no impact on the correctness or the termination of the algorithm, so that the instructions at lines 3 and 16 could be frozen as well (but again using a cache proves to be computationally useful in many cases).

Empirical Evaluation

In order to evaluate the benefits offered by projMC, we performed some experiments. The empirical protocol we followed is precisely the same as the one considered in (Aziz et al. 2015). Thus we have considered 260 instances coming from three data sets. The first data set consists of 100 instances, based on random 3-CNF formulae, where the number of variables is set to 100 and the number of clauses is varied. Clause-to-variable ratios of 1, 1.5, 2, 3, and 4 have been considered. We let also the number of variables of X to vary (the elements of X are chosen uniformly at random). The second data set consists of 60 instances, corresponding to random Boolean circuits based on 30 variables. Those circuits are generated as follows. One keeps a set containing at start the 30 variables, and as long as the set is not a singleton, we randomly select an operator o (AND, OR, NOT), pick up operands V for o in the set at random, create a new variable v, add the gate $o \leftrightarrow o(V)$ to the circuit, and put v back in the set. The process is repeated c times, with c equals

to 1, 5, or 10. Finally, the third data set consists of 100 instances generated from five classical planning problems (depots, driver, rovers, logistics, and storage) considered under varying planning horizons. For each problem and value of the horizon, two variants are considered, one with the goal state fixed and one where the goal is relaxed to be any viable goal. For the first variant, the projected model count to be computed represents the number of initial states the given plan can achieve the goal from. For the second variant, it gives the number of initial states plus all goal configurations that the given plan works for.

For each instance, we measured the time (in seconds) required by projMC to achieve the projected model counting job. In the experiments, the model counter MC used in projMC is the top-down compilation-based model counter D4 described in (Lagniez and Marquis 2017). For the sake of comparison, we have also run the previous projected model counters dSharp$_P$, #clasp, and d2c on the same instances (those solvers are available from people.eng.unimelb.edu.au/pstuckey/countexists) and measured the corresponding computation times. In addition, we have also compared projMC with D4$_P$, which is the same algorithm as dSharp$_P$, but using D4 as the underlying model counter instead of dSharp. All the experiments have been conducted on a cluster of Intel Xeon E5-2643 (3.30 GHz) quad core processors with 32 GiB RAM. The kernel used was CentOS 7, Linux version 3.10.0-514.16.1.el7.x86_64. The compiler used was gcc version 5.3.1. Hyperthreading was disabled, and no memory share between cores was allowed. A timeout of 600s and a memory-out of 7.6 GiB has been considered for each instance.

The results are reported on the scatter plots given in Figure 1. Each dot represents an instance; the time (in seconds) needed to solve it using the projected model counter corresponding to the x-axis (resp. y-axis), is given by its x-coordinate (resp. y-coordinate). Logarithmic scales are used for both coordinates. In part (a) (resp. (b), (c), (d)) of the figure, the x-axis corresponds to dSharp$_P$ (resp. #clasp, d2c, D4$_P$). The y-axis corresponds to projMC in each part of the figure.

The four scatter plots in Figure 1 clearly show that for a great majority of instances the time needed by projMC to count the number of projected models is smaller (and often significantly smaller) than the corresponding computation times when the other projected model counters are used. This is especially the case when the "trivial" instances (i.e., those solved with a second or alike) are neglected. Notwithstanding those instances, it is interesting to observe that projMC appears at least as efficient as any of the other projected model counters for all the instances from the random and the planning data sets.

The cactus plot in Figure 2 gives for dSharp$_P$, #clasp, d2c, D4$_P$, and projMC the number of instances solved in a given amount of time. Clearly enough, projMC outperforms the previous projected model counters: when the "trivial" instances have been discarded, projMC typically solves more instances than any of them in any given amount of time. Especially, some significant benefits in terms of the number of instances solved have been obtained. Thus, Table 1 makes precise for each projected model counter under consideration the number of instances (over 260) which have been solved within the time limit of 600s. It can be observed that projMC has been able to solve many more instances than the other projected model counters.

projected model counter	# of instances solved
dSharp$_P$	115
#clasp	94
d2c	71
D4$_P$	140
projMC	192

Table 1: Number of instances solved within the time limit depending on the projected model counter used.

Table 2 reports for each projected model counter under consideration the number of instances which have been solved by it, and only by it.

projected model counter	# of instances uniquely solved
dSharp$_P$	0
#clasp	1
d2c	0
D4$_P$	1
projMC	44

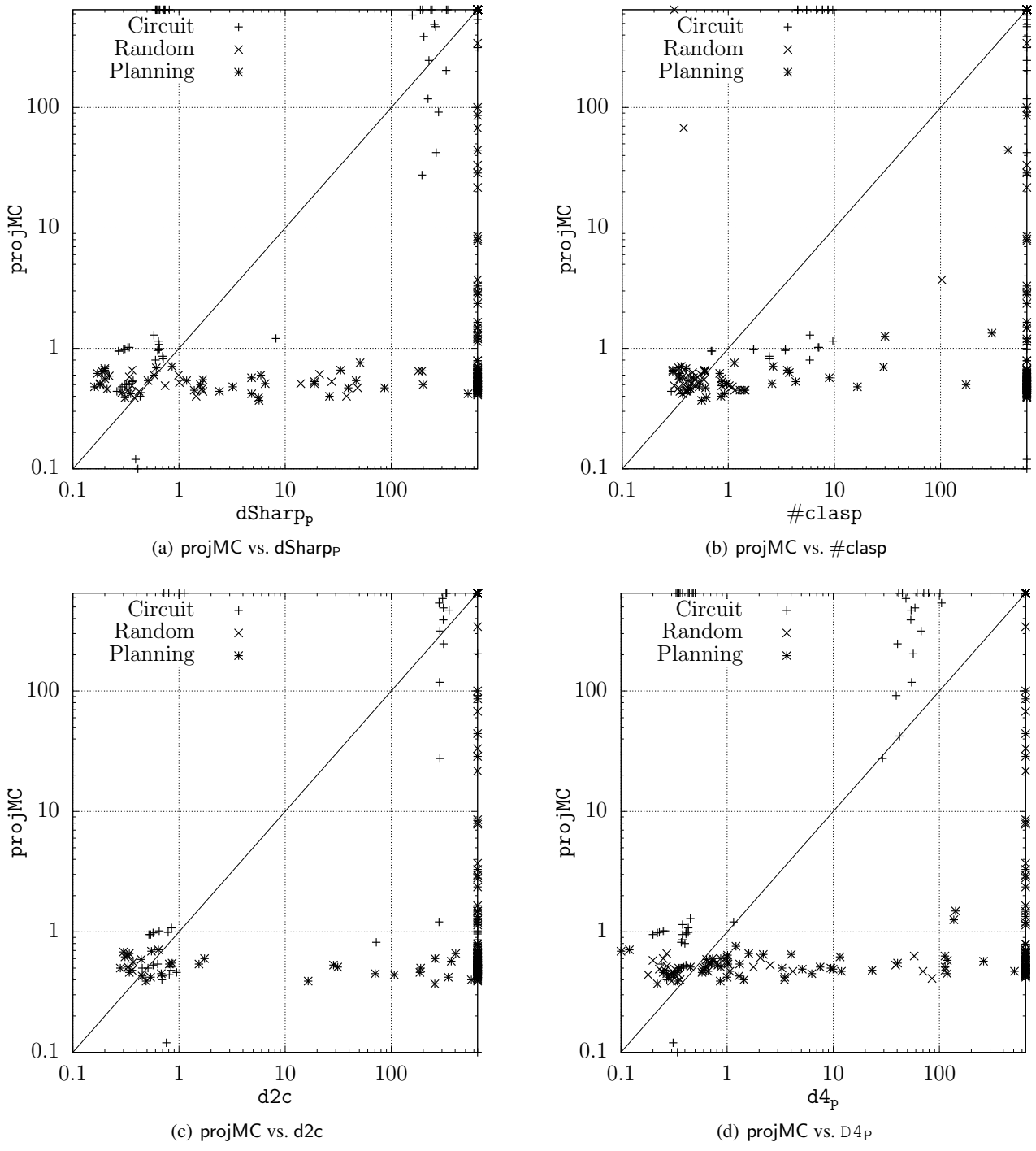
Table 2: Number of instances uniquely solved within the time limit depending on the projected model counter used.

Table 2 shows that projMC was able to solve a significant number of instances that were out of reach for the other projected model counters, given the time limit under consideration. That mentioned, the virtual best solver for our experiments would solve 211 instances, which is slightly above 192. This coheres with the results reported in Figure 1 (d), showing that for the circuit instances, D4$_P$ typically challenges projMC.

Conclusion and Perspectives

We have presented a new algorithm, projMC, for computing the number of models $\|\exists X.\Sigma\|$ of a propositional formula Σ after eliminating from it a given set X of variables. Unlike previous algorithms, projMC takes advantage of a disjunctive decomposition scheme of $\exists X.\Sigma$ for computing $\|\exists X.\Sigma\|$. It also looks for disjoint components in its input for improving the computation. Our experiments have shown that projMC can be significantly more efficient than the existing algorithms dSharp$_P$, #clasp, and d2c for projected model counting. Empirically, projMC also proved better than D4$_P$ on many instances, showing that the improved performance of projMC is not solely due to the the fact that it is "powered" by D4.

A first perspective for further research consists in turning our projected model counter into a compiler generating a d-DNNF representation from a CNF formula containing



(a) projMC vs. dSharp$_P$

(b) projMC vs. #clasp

(c) projMC vs. d2c

(d) projMC vs. D4$_P$

Figure 1: Comparing projMC with dSharp$_P$, #clasp, d2c, and D4$_P$. The coordinates correspond to computation times in seconds. Logarithmic scales are used.

some existentially quantified variables. Provided that the underlying model counter MC is replaced by a d-DNNF compiler (like C2D (Darwiche 2002; 2004), dSharp (Muise et al. 2012) or D4 (Lagniez and Marquis 2017)), the changes to be done mainly consist in modifying the instructions at lines 14 and 15 of Algorithm 1 to generate a deterministic OR node instead of making a summation. In such a compiler, when a model of Σ is generated (i.e., at a step corresponding to line 10 of Algorithm 1), one could look for an assignment maximizing the number of clauses which are satisfied by

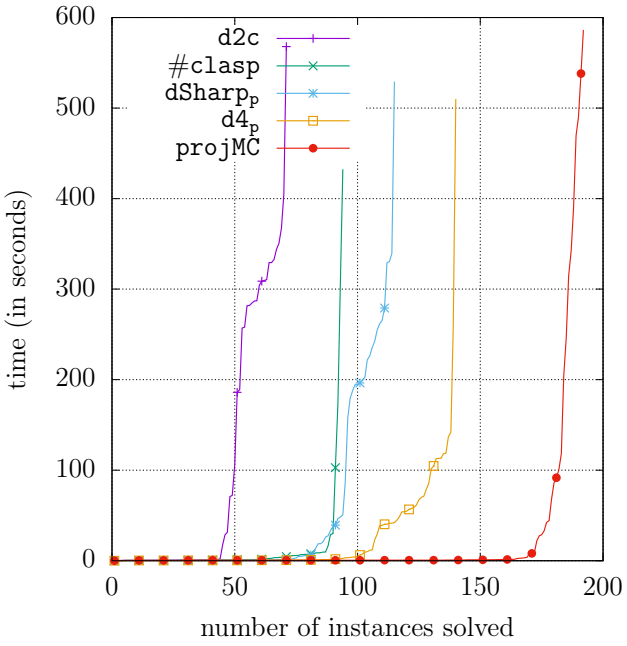


Figure 2: Number of instances solved by dSharp$_P$, #clasp, d2c, D4$_P$, and projMC in a given amount of time.

some existentially quantified literals of the model, so as to get a deterministic disjunctive form which contains as few elements as possible (hence a deterministic OR node with as few children as possible). In fact, one already tested this approach within projMC but the benefits achieved are not that significant in this case – the time spent in the maximisation processes can be quite large. However, things can be different when the objective is to generate a compiled representation since, in such a setting, one is typically ready to spend more off-line time in the compilation step, provided that the size of the associated compiled form is significantly smaller. We plan to make some experiments in this direction to determine whether this approach could prove useful for generating more succinct compiled representations.

A second perspective will consist in taking advantage of the two programs B and E for gate detection and replacement within CNF formulae, used as the key components of the preprocessor for model counting reported in (Lagniez, Lonca, and Marquis 2016). Indeed, B and E could be exploited as additional inprocessing filtering techniques in projMC. It would be interesting to determine whether this could be computationally useful, especially for solving circuit instances where, by construction, many gates can be found.

A last perspective will consist in evaluating projMC and the other projected model counters on other benchmarks, especially those reported in the repository available from https://github.com/dfremont/counting-benchmarks/tree/master/benchmarks/projection.[1]

[1]We would like to thank an anonymous reviewer for pointing out this dataset.

References

Aziz, R. A.; Chu, G.; Muise, C. J.; and Stuckey, P. J. 2015. #∃SAT: Projected model counting. In *Proc. of SAT'15*, 121–137.

Castell, T. 1996. Computation of prime implicates and prime implicants by a variant of the Davis and Putnam procedure. In *Proc. of ICTAI'96*, 428–429.

Darwiche, A. 2002. A compiler for deterministic decomposable negation normal form. In *AAAI'02*, 627–634.

Darwiche, A. 2004. New advances in compiling CNF into decomposable negation normal form. In *Proc. of ECAI'04*, 328–332.

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

Fichte, J. K.; Hecher, M.; Morak, M.; and Woltran, S. 2018. Exploiting treewidth for projected model counting and its limits. In *Proc. of SAT'18*, 165–184.

Gebser, M.; Kaufmann, B.; and Schaub, T. 2009. Solution enumeration for projected Boolean search problems. In *Proc. of CPAIOR'09*, 71–86.

Hyvärinen, A. E. J.; Junttila, T. A.; and Niemelä, I. 2006. A distribution method for solving SAT in grids. In *Proc. of SAT'06*, 430–435.

Klebanov, V.; Manthey, N.; and Muise, C. J. 2013. SAT-based analysis and quantification of information flow in programs. In *Proc. of QUEST'13*, 177–192.

Lagniez, J.-M., and Marquis, P. 2017. An improved decision-DNNF compiler. In *Proc. of IJCAI'17*, 667–673.

Lagniez, J.-M.; Lonca, E.; and Marquis, P. 2016. Improving model counting by leveraging definability. In *Proc. of IJCAI'16*, 751–757.

Lang, J.; Liberatore, P.; and Marquis, P. 2003. Propositional independence: Formula-variable independence and forgetting. *Journal of Artificial Intelligence Research* 18:391–443.

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

Muise, C.; McIlraith, S.; Beck, J.; and Hsu, E. 2012. Dsharp: Fast d-DNNF compilation with sharpSAT. In *Proc. of AI'12*, 356–361.

Schrag, R. 1996. Compilation for critically constrained knowledge bases. In *Proc. of AAAI'96*, 510–515.

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

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

Zhang, H., and Stickel, M. 1996. An efficient algorithm for unit propagation. In *Proc. of ISAIM'96*, 166–169.