

Algorithms for Projected Knowledge Compilation

Anonymous Authors

No Institution Given

Abstract. Knowledge compilers convert Boolean formulas, given in conjunctive normal form (CNF), into representations that enable efficient evaluation of unweighted and weighted model counts, as well as a variety of other useful properties. Often, however, CNF formulas contain auxiliary variables either to enable efficient encoding or to enable composition of multiple components. The output of a standard knowledge compiler will include these variables, making it difficult to compute properties of the underlying formula. In this paper, we present and evaluate five algorithms for *projected knowledge compilation*, where auxiliary variables are eliminated by existential quantification. Supporting projection greatly expands the range of applications for knowledge compilation.

1 Introduction

Although Boolean satisfiability (SAT) solvers serve as effective tools for many automated reasoning tasks, some applications require evaluating more detailed properties about formulas than whether or not they are satisfiable. For example, *model counting* [14] determines the number of satisfying assignments to a formula, while *probabilistic inference* [5] determines the probability that a formula will evaluate to true, given individual probabilities that each variable is assigned value 1. Probabilistic inference is a special case of the more general problem of *weighted model counting*. Generalizing to weight functions computed over semirings expands properties that can be evaluated even further [18].

Rather than implementing specialized programs for these different analysis tasks, an approach known as *knowledge compilation* [9,19] converts a Boolean formula into a form for which the tasks can be performed in polynomial time, relative to the size of the representation. Most commonly, knowledge compilers generate a restricted class of Boolean formulas for which the analysis becomes tractable. These formulas are encoded as directed acyclic graphs, enabling efficient sharing of common subformulas.

Many applications introduce auxiliary variables when encoding formulas into the conjunctive normal form (CNF) required by most SAT solvers and knowledge compilers. For example, consider a *majority* formula $Maj(X)$ that evaluates to true when more than $n/2$ of the n variables in X are assigned 1. Encoding this directly into CNF requires an exponential number of clauses [29]. On the other hand, by introducing $O(n^2)$ encoding variables Z , we can encode a formula $Maj(X, Z)$, having $O(n^2)$ clauses, such that, when a SAT solver finds a solution to $Maj(X, Z)$, the values assigned to the variables in X will serve as a satisfying assignment for $Maj(X)$ [29]. We write this as $Maj(X) \equiv \exists Z Maj(X, Z)$.

Other applications introduce auxiliary variables to enable the composition of multiple components. For example, a key step in symbolic model checking [4] requires composing a transition relation $T(X, X')$ describing possible pairs of current (over variables X) and next states (over variables X') with a predicate $R_i(X)$ indicating the set of states that can be reached by a sequence of i transitions from an initial set of states. The set of states reachable in $i + 1$ transitions is then given by the formula $R_{i+1}(X') = \exists X [R_i(X) \wedge T(X, X')]$, a structure known as a *relational product* [3]. Although knowledge compilers have not historically been used for symbolic model checking, the ability to compile relational products would make this possible.

Unfortunately, introducing auxiliary variables complicates the analyses enabled by knowledge compilation. The output of a standard knowledge compiler will include these variables in the output formula, with the possibility that multiple models of the generated formula all map to a single assignment for the data variables.

This paper establishes a framework for *projected knowledge compilation*. Given Boolean formula $\phi(D, P)$ over sets of *data* variables D and *projection* variables P , such a compiler generates a representation of the formula $\exists P \phi(D, P)$ in a form suitable for the same analysis tasks as standard knowledge compilation.

We identify the special role that *Tseitin* variables can serve in projected knowledge compilation. These are projection variables with values uniquely determined by the values assigned to the data variables. They occur, for example, when the projection variables serve only to enable a Tseitin encoding of Boolean operations [28]. Such variables can be more readily removed than other types of projection variables. Indeed, we show that compiler performance can be improved by adding *blocked clauses* [16,17] to the input formula to “promote” other projection variables into Tseitin variables.

We present five algorithms for projected knowledge compilation. Several use a reduction we refer to as *trimming*, where each occurrence of a projection variable or its complement in a formula is replaced with tautology. The TRIM method simply trims the output of a standard knowledge compiler. It is suitable only when all projection variables are Tseitin variables.

The SPLIT-D (for ‘Split Data’) method extends an algorithm developed for projected model counting [1]. It is implemented in the most recent version of the D4 knowledge compiler,¹ to our knowledge, the only prior implementation of a projected knowledge compiler. This method constrains the knowledge compiler to only split on data variables until the subformula contains only projection variables. At that point, a SAT solver can determine whether the subformula is satisfiable or not, yielding counts 1 and 0, respectively. The SPLIT-DT method extends SPLIT-D by running a preprocessor to detect (and possibly promote) Tseitin variables and then grouping them with data variables during splitting.

The RECOMPILE algorithm also extends an algorithm developed for projected model counting [1]. It starts by trimming the output of a standard knowledge compiler to remove all projection variables. As we will show, the resulting formula is a valid representation of $\exists P \phi(D, P)$, but it may violate the restrictions that enable efficient analysis. However, it can be converted into clausal form using Tseitin encodings of the sum and

¹ Available at <https://github.com/crillab/d4v2>

product operations, recompiled with the knowledge compiler, and then have the encoding variables removed via trimming.

Finally, the REPAIR method modifies the output of a standard knowledge compiler, trimming literals and introducing intermediate terms in the formula to ensure that, for each Boolean sum, overlapping assignments for the arguments are only counted once. In doing so, REPAIR introduces logical negations into the compiled formula. Even with negations, computing key properties, including both weighted and unweighted model counts, remains tractable.

We have implemented a projected knowledge compiler PKC that supports all five compilation algorithms. We evaluate PKC on multiple encodings of majority formulas, as well as on benchmarks from recent projected model counting competitions. We find that each algorithm has unique strengths and weaknesses.

2 Prior Work

A number of knowledge compilers have been developed over the years. One successful class is based on algorithms inspired by conflict-directed, clause-learning (CDCL) SAT solvers [8,21,25]. That is, they operate top-down, combining unit propagation with recursive splitting on variables. They learn clauses to encode portions of the solution space that are unsatisfiable. Unlike a SAT solver, the knowledge compiler does not stop when it finds a satisfying solution but instead explores the entire solution space. The results are expressed as formulas (represented by directed acyclic graphs) in a restricted form known as *decision-decomposable, negation normal form* (decision-DNNF), defined in Section 3.2. Our algorithms make use of this type of compiler.

Aziz, et al., introduced the *projected model counting* problem, with the objective of computing the number of models of $\exists P \phi(D, P)$, and they describe three different algorithms [1]. The first and third of these we adapted to form projected compilation algorithms SPLIT-D and RECOMPILE. Their second approach maintains a dual representation of the partial solutions, with conjunctive representations of subformulas and disjunctive representations of the sets of data variable assignments covered. This approach was refined by Lagniez and Marquis in their tool PROJMC [22]. That paper mentions that a dual approach could be generalized to implement a projected knowledge compiler, but, to the best of our knowledge, such a tool has never been developed.

Dudek extended their model counter based on algebraic decision diagram [10] to support projected model counting [11]. Their approach is similar to the SPLIT-D method: they order the variables to have the projection variables below the data variables, and so the projection variables can be eliminated by existential quantification.

3 Logical Foundations

Let X denote a set of Boolean variables, and let ρ be an *assignment* of truth values to some subset of the variables, where 0 denotes false and 1 denotes true, i.e., $\rho: Y \rightarrow \{0, 1\}$ for some $Y \subseteq X$. We say the assignment is *total with respect to a set of variables* Y , when it assigns a value to every $x \in Y$. The set of all possible total assignments with respect to a set of variables Y is denoted $\mathcal{U}(Y)$.

For each variable $x \in X$, we define *literals* x and \bar{x} , where \bar{x} is the negation of x . An assignment ρ can be viewed as a set of literals, with $x \in \rho$ when $\rho(x) = 1$, and $\bar{x} \in \rho$ when $\rho(x) = 0$. We write the negation of literal ℓ as $\bar{\ell}$. That is, $\bar{\ell} = \bar{x}$ when $\ell = x$ and $\bar{\ell} = x$ when $\ell = \bar{x}$. The variable associated with literal ℓ is denoted $\text{var}(\ell)$. That is $\text{var}(x) = \text{var}(\bar{x}) = x$. Total assignment ρ is said to be *consistent* with partial assignment ρ' when $\rho' \subseteq \rho$.

3.1 Boolean Formulas and their Encodings into CNF

The set of Boolean formulas is defined recursively:

- Constants: \perp (logical falsehood) and \top (tautology)
- Literals: x and \bar{x} for $x \in X$
- Product operations: $\bigwedge_{1 \leq i \leq k} \psi_i$
- Sum operations: $\bigvee_{1 \leq i \leq k} \psi_i$
- Negation operations: $\neg\psi$

Boolean formula ψ has an associated *dependency set* $\mathcal{V}(\psi) \subseteq X$, consisting on those variables that occur in ψ . It also has a set of *models* $\mathcal{M}(\psi) \subseteq \mathcal{U}(X)$, consisting of those total assignments for which the formula evaluates to true.

We say that formulas ψ and ϕ are *equivalent*, written $\psi \equiv \phi$ when $\mathcal{M}(\psi) = \mathcal{M}(\phi)$. Formula ψ is said to *entail* formula ϕ , written $\psi \models \phi$, when $\mathcal{M}(\psi) \subseteq \mathcal{M}(\phi)$.

A formula is said to be in *negation-normal form* (NNF) when it contains no negation operations. A formula is said to be in *conjunctive normal form* (CNF) when (i) it is in negation-normal form, (ii) the arguments of all sum operations are literals, and (iii) there is a single product operation having all sum operations as its arguments. The sum operations are referred to as *clauses*. When discussing CNF, it is common to use set notation: the clauses are sets of literals, and the formula is a set of clauses.

Standard Boolean identities can be used to reduce the size of a formula. We will say that a formula has been *simplified* when it satisfies the following: (i) it contains constant \perp or \top only if the formula itself is constant, (ii) negation operations do not have literals or other negation operations as arguments, and (iii) every product and sum operation has at least two arguments.

We use two common methods for encoding a Boolean formula into CNF. Assume first that the formula has been simplified and is not constant, and so it consists only of literals, as well as sum, product, and negation operations. For subformula $\psi = \psi_1 \text{ op } \psi_2$, where $\text{op} \in \{\wedge, \vee\}$, we introduce an auxiliary variable v , and v becomes the *associated literal* for ψ . The associated literal for literal ℓ is ℓ itself, while the associated literal for $\neg\psi_1$ is $\bar{\ell}_1$, where ℓ_1 is the literal associated with ψ_1 . Let ℓ_1 and ℓ_2 be the literals associated with arguments ψ_1 and ψ_2 . The *Tseitin* encoding [28] adds defining clauses enforcing $v \leftrightarrow \ell_1 \text{ op } \ell_2$. It can be used to encode an arbitrary Boolean formula. The defining clauses in the *Plaisted-Greenbaum* encoding [27] enforce only one side of this equivalence: $v \rightarrow \ell_1 \text{ op } \ell_2$. It can only be applied to NNF formulas. The extension from operations with two arguments to more is straightforward. To complete the encoding of top-level formula ψ with associated literal ℓ , unit clause $[\ell]$ is added, asserting that ψ must evaluate to true. As we will see, the choice of encoding has significant impact on how easily a projected knowledge compiler can remove the auxiliary variables.

3.2 Restricted Formula Classes and (Weighted) Model Counting

Model counting and other analysis tasks become tractable when we enforce a set of restrictions on a formula. A Boolean product $\bigwedge_{1 \leq i \leq k} \psi_i$ is said to be *decomposable* [6,9] when the subformulas are defined over disjoint sets of variables: $\mathcal{V}(\psi_i) \cap \mathcal{V}(\psi_j) = \emptyset$ for all $1 \leq i < j \leq k$. A Boolean sum $\bigvee_{1 \leq i \leq k} \psi_i$ is said to be *deterministic* [7,9] when the models of its subformulas are disjoint: $\mathcal{M}(\psi_i) \cap \mathcal{M}(\psi_j) = \emptyset$ for all $1 \leq i < j \leq k$. A Boolean formula is said to be deterministic and decomposable (DD) when every product operation is decomposable, and every sum operation is deterministic [23]. A formula is said to be in deterministic, decomposable negation-normal form (DDNNF) when it is both in negation-normal form, and it is DD.

As an important subclass, a DDNNF formula is said to be in *decision*, decomposable negation-normal form (decision-DNNF) when every sum operation is of the form $\psi_1 \vee \psi_2$, and it has an associated variable $x \in X$, such that one argument restricts all satisfying assignments ρ to have $\rho(x) = 1$, while the other restricts them to have $\rho(x) = 0$ [26]. More precisely, for some $i \in \{1, 2\}$, either $\psi_i = x$ or ψ_i is a product operation having x as an argument, and conversely, letting $j = 3 - i$, either $\psi_j = \bar{x}$ or ψ_j is a product operation having \bar{x} as an argument. Our algorithms make use of knowledge compilers that generate formulas in decision-DNNF form.

For Boolean formula ψ , the *normalized*, weighted model counting problem starts by associating a real-valued weight $w(x)$ with every $x \in X$, such that $0 \leq w(x) \leq 1$. The weight for literal \bar{x} is then defined as $w(\bar{x}) = 1 - w(x)$. The weighted model count of ψ is then defined as $\mathbf{W}[\psi, w] = \sum_{\alpha \in \mathcal{M}_{\mathcal{D}}(\psi)} \prod_{\ell \in \alpha} w(\ell)$. That is, the weight of an assignment is the product of the weights of the literals, and the weighted model count of a formula is the sum of the weights of its models. When formula ψ is DD, its weighted count can be computed recursively:

- Constants: $\mathbf{W}[\top] = 1$, and $\mathbf{W}[\perp] = 0$.
- Literals: $\mathbf{W}[x] = w(x)$ and $\mathbf{W}[\bar{x}] = 1 - w(x)$.
- Product operations: $\mathbf{W}\left[\bigwedge_{1 \leq i \leq k} \psi_i\right] = \prod_{1 \leq i \leq k} \mathbf{W}(\psi_i)$
- Sum Operations: $\mathbf{W}\left[\bigvee_{1 \leq i \leq k} \psi_i\right] = \sum_{1 \leq i \leq k} \mathbf{W}(\psi_i)$
- Negations: $\mathbf{W}[\neg\psi] = 1 - \mathbf{W}[\psi]$

Having the product operations be decomposable and the sum operations be deterministic avoids the difficulties of replacing the idempotent Boolean product and sum operations by arithmetic multiplication and addition.

The more general weighted model counting problem allows independently choosing real-valued weights $W(\ell)$ for each literal ℓ , with the possibility that $W(x) + W(\bar{x}) \neq 1$ for some $x \in X$. As long as the two weights for each variable do not sum to 0, this more general form can be computed by defining $R(x) = W(x) + W(\bar{x})$ and normalizing the weights as $w(x) = W(x)/R(x)$ and $w(\bar{x}) = W(\bar{x})/R(x)$. Scaling $\mathbf{W}[\psi, w]$ by $\prod_{x \in D} R(x)$ yields the weighted count. An (unweighted) count of the models for ψ can be computed using weights $W(x) = W(\bar{x}) = 1$ for every variable $x \in X$. That is, weighted evaluation is performed with $w(x) = w(\bar{x}) = 1/2$ for every variable x , and the result is scaled by 2^n , where $n = |X|$.

Most research on knowledge compilation has focused on negation-normal forms, especially DDNNF [9]. Generalizing the weight functions to computations over semirings enables efficient computation of many properties of DDNNF formulas [18]. By contrast, our REPAIR method generates formulas that are DD but contain negation operators, and hence they are not DDNNF. Properties that can be encoded as weight functions over rings,² including (weighted) model counting, can be computed efficiently for arbitrary DD formulas.

3.3 Projection

With projected formulas, the set of Boolean variables X is divided into two sets: *data* variables D and *projection* variables P . We use separate assignments for the two sets of variables: $\alpha: D \rightarrow \{0, 1\}$ and $\beta: P \rightarrow \{0, 1\}$ and the symbol \cdot to combine the two assignments: $\alpha \cdot \beta: X \rightarrow \{0, 1\}$. Sets $\mathcal{U}(D)$ and $\mathcal{U}(P)$ therefore denote the set of all assignments that are total with respect to D and P , respectively.

For Boolean formula ψ , its set of *data models*, written $\mathcal{M}_{\mathcal{D}}(\psi)$ is defined as:

$$\mathcal{M}_{\mathcal{D}}(\psi) = \{\alpha \in \mathcal{U}(D) \mid \exists \beta \in \mathcal{U}(P) \alpha \cdot \beta \in \mathcal{M}(\psi)\} \quad (1)$$

Of particular interest among projection variables are those that have unique assignments with respect to the data variables. For formula ψ , a projection variable $z \in P$ is classified as a *Tseitin variable* when for every data variable assignment $\alpha \in \mathcal{U}(D)$ and every pair of projection variable assignments $\beta_1, \beta_2 \in \mathcal{U}(P)$ such that both $\alpha \cdot \beta_1$ and $\alpha \cdot \beta_2$ are in $\mathcal{M}(\psi)$, the projection variable assignments satisfy $\beta_1(z) = \beta_2(z)$. In general, this condition requires that variable z , as well as all projection variables on which z (transitively) depends, be Tseitin encodings of Boolean operations.

Tseitin variables, even when projected out, impose an important condition on the set of data models for a formula. As notation, for literal ℓ such that $\text{var}(\ell) \in P$, define

$$\mathcal{M}_{\mathcal{D}}^{\ell}(\psi) = \{\alpha \in \mathcal{U}(D) \mid \exists \beta \in \mathcal{U}(P) \alpha \cdot \beta \in \mathcal{M}(\psi) \text{ and } \ell \in \beta\} \quad (2)$$

Proposition 1. *Tseitin variable z partitions the set of data models for Boolean formula ψ into disjoint subsets: $\mathcal{M}_{\mathcal{D}}^{\bar{z}}(\psi) \cap \mathcal{M}_{\mathcal{D}}^z(\psi) = \emptyset$.*

This property follows from the definitions: having data variable assignment α be in both sets would require projection variable assignments β_1 and β_2 such that both $\alpha \cdot \beta_1$ and $\alpha \cdot \beta_2$ are in $\mathcal{M}(\psi)$, but $\beta_1(z) = 0$ while $\beta_2(z) = 1$.

4 Projection Algorithms

Projected knowledge compilation can be defined as follows: given CNF formula ϕ , generate a DD formula ψ having $\mathcal{V}(\psi) \subseteq D$, such that $\mathcal{M}(\psi) = \mathcal{M}_{\mathcal{D}}(\phi)$. We write this as $\psi \equiv \exists P \phi$. Here we describe five approaches, as summarized in Table 1. As is indicated, TRIM only applies to a limited class of formulas, while the others are general purpose.

² A ring requires every element to have an additive inverse; a semiring does not.

Table 1. Properties of the projected knowledge compilation algorithms

Method	Restrictions	Output Form	KC Invocations
TRIM	Tseitin variables	DDNNF	1
SPLIT-D	None	decision-DNNF	1
SPLIT-DT	None	DDNNF	1
RECOMPILE	None	DDNNF	2
REPAIR	None	DD	1–many

Each provides different restrictions on the output format, with decision-DNNF being the most restrictive and DD being the least. All use a standard knowledge compiler, with the number of invocations depending on the method and the input formula.

As a preprocessing step, we can perform standard Boolean constraint propagation on ϕ [24]. In anticipation of eliminating projection variables, we can also perform a form of *pure literal* elimination as follows. For NNF formula ψ , we say that literal ℓ such that $\text{var}(\ell) \in P$ is *pure* when ℓ occurs in ψ , but there are no occurrences of $\bar{\ell}$. Pure literal elimination involves replacing all occurrences of pure literal ℓ with tautology and simplifying the result. This rule is justified as follows:

Proposition 2. *If literal ℓ is pure in NNF formula ψ , then $\mathcal{M}_{\mathcal{D}}^{\bar{\ell}}(\psi) \subseteq \mathcal{M}_{\mathcal{D}}^{\ell}(\psi)$, and therefore $\mathcal{M}_{\mathcal{D}}(\psi) = \mathcal{M}_{\mathcal{D}}^{\ell}(\psi)$.*

This proposition can be proved by inducting on the structure of ψ . Applying pure literal elimination to a CNF formula can cause other literals to become pure, and so the rule should be iterated until no new pure literals remain.

We can apply a preprocessing phase to heuristically detect Tseitin variables in the CNF formula ϕ , as is done by some preprocessors for model counters [20] and by SAT solvers to enable bounded variable elimination [12,13,15]. The preprocessor detects Tseitin variables starting with those that depend only on data variables and adding those that depend only on data variables and previously detected Tseitin variables. At each step, it identifies a set of clauses $\theta \subseteq \phi$ such that dependency set $\mathcal{V}(\theta)$ contains only a candidate variable z , plus data and previously detected Tseitin variables. It then modifies these clauses by removing all instances of z and \bar{z} , obtaining a set of clauses θ' . If θ' is unsatisfiable, then the values assigned to data and other Tseitin variables uniquely determine the value of z , and therefore z is a Tseitin variable.

The preprocessor can also perform *Tseitin promotion*, modifying the input formula to convert other projection variables into Tseitin variables without changing the set of data models. For input formula ϕ and projection variable z , let ℓ be either z or \bar{z} , and let $\phi_{\ell} \subseteq \phi$ denote the set of all clauses that contain ℓ . We can promote z under the following conditions: (i) dependency set $\mathcal{V}(\phi_{\ell})$ contains only data variables, previously detected Tseitin variables, and z , (ii) there are no complementary literals in ϕ_{ℓ} , i.e., when both literals x and \bar{x} are present for some variable x . Let $\phi_{\ell} = \{C_1, C_2, \dots, C_k\}$. We promote z by adding all clauses B of the form $B = [\bar{\ell} \vee \bar{\ell}_1 \vee \bar{\ell}_2 \vee \dots \vee \bar{\ell}_k]$, where each $\ell_i \in C_i$ and $\ell_i \neq \ell$ for $1 \leq i \leq k$. These clauses are *blocked* [16,17] and so will not reduce the set of data models. Furthermore they guarantee that the value of z

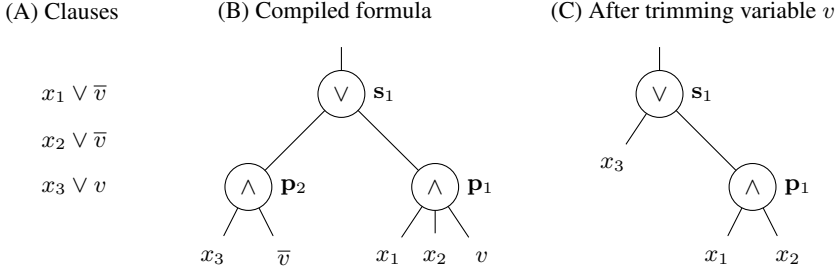


Fig. 1. Example of knowledge compilation and trimming for Boolean formula $x_3 \vee (x_1 \wedge x_2)$

is uniquely determined by the assignments to the data variables. Proofs of these two properties are presented in Appendix A.2.

For CNF formula ϕ , we define $\text{KC}(\phi)$ as the result of applying a knowledge compiler to ϕ to generate decision-DNNF formula ψ such that $\phi \equiv \psi$. We also define $\text{Clausify}(\psi)$ to be an operation that converts Boolean formula ψ into CNF, using a Tseitin encoding of the Boolean operations, as described in Section 3.1. It generates both a CNF formula ϕ and a set of auxiliary variables Z , all of which are Tseitin variables.

4.1 Trimming

The first algorithm applies directly only to formulas where all projection variables are Tseitin variables, but it also serves as an important step in the other methods. For Boolean formula ψ , we define $\text{Trim}(\psi, V)$ for $V \subseteq X$ as the formula obtained by replacing every literal ℓ in ψ for which $\text{var}(\ell) \in V$ by \top and then simplifying the result.

Proposition 3. *For NNF formula ψ , where all of its product operations are decomposable, $\text{Trim}(\psi, P) \equiv \exists P \psi$.*

This result can be proved by induction on the structure of ψ . The key properties are that existential quantification commutes with sum operations and with decomposable product operations. Letting $\theta = \text{Trim}(\psi, P)$, we can also see that each subformula θ' in θ will have a counterpart ψ' in ψ such that $\theta' = \text{Trim}(\psi', P)$, and therefore $\theta' \equiv \exists P \psi'$.

Trimming suggests the following approach to projected knowledge compilation. Starting with CNF formula ϕ , apply a standard knowledge compiler to generate $\psi = \text{KC}(\phi)$, and then generate $\theta = \text{Trim}(\psi, P)$. The result will satisfy $\theta \equiv \exists P \phi$. Moreover, it will be in negation-normal form, and its product operations will all be decomposable. In general, however, θ will contain sum operations that are not deterministic.

As an example, Figure 1 shows the effect of compiling and trimming a set of clauses (A). These provide a CNF encoding of $x_3 \vee (x_1 \wedge x_2)$ using a Plaisted-Greenbaum encoding of subformula $x_1 \wedge x_2$ with auxiliary variable v . In (B), we can see that the knowledge compiler split on variable v at the top level, giving x_3 when v is assigned 0 and $x_1 \wedge x_2$ when v is assigned 1. This formula is trimmed (C) by replacing both literals \bar{v} and v with \top . Product p_1 then simplifies to $x_1 \wedge x_2$, and p_2 simplifies to x_3 .

As can be seen, trimming gives exactly the formula we are trying to encode, but the sum operation s_1 now violates determinism — both arguments will be satisfied when x_1 , x_2 , and x_3 are all assigned 1.

Some important cases guarantee that at least some of the sum operation in θ will be deterministic. Consider sum operation $\theta' = \theta_1 \vee \theta_2$ in θ and its counterpart $\psi' = \psi_1 \vee \psi_2$ in ψ . Assume that ψ' has an associated decision variable x . Then $\mathcal{M}(\theta_1) \cap \mathcal{M}(\theta_2) = \emptyset$ when x is either a data variable or a Tseitin variable. In the former case, literals x and \bar{x} are unaffected by trimming, and so x will be a decision variable for θ' . The latter case follows directly by Proposition 1. Observe that the sum operations generated by trimming Tseitin variables will be deterministic, but they will not have associated decision variables. The resulting formula will therefore be DDNNF but perhaps not decision-DNNF.

This result shows that, when all projection variables are Tseitin variables, the output of a knowledge compiler can be transformed via trimming to be suitable as the output of a projected knowledge compiler. This is an important result for application developers who can choose how they encode problems into conjunctive normal form. By using Tseitin encodings for all operations, a simple trimming will suffice to eliminate them. We will find this property useful in our other methods.

4.2 The SPLIT-D and SPLIT-DT Methods

Two other approaches to projected knowledge compilation can be seen as optimizations of knowledge compilation followed by trimming. Both involve running the knowledge compiler on CNF formula ϕ to generate decision-DNNF formula $\psi = \text{KC}(\phi)$. However, the SPLIT-D method constrains the knowledge compiler to choose a data variable as the splitting variable whenever possible. It will therefore split on a projection variable only when the set of reduced clauses contains only projection variables. Any sum operation ψ' in ψ having an associated decision variable $x \in P$ will therefore have $\mathcal{V}(\psi') \subseteq P$. Trimming ψ' will yield (through simplifications) constant \top . As a result, all sum operations in $\theta = \text{Trim}(\psi, P)$ will have data variables as decision variables, and θ will also be decision-DNNF.

The SPLIT-DT method works similarly, except that it allows splitting on both data variables and Tseitin variables until none remain. The Tseitin variables are removed by trimming the generated result. That is, we compute $\text{PKC}(\phi, P)$ as $\text{Trim}(\text{PKC}(\phi, P - Z), Z)$, where $Z \subseteq P$ are the Tseitin variables detected (or created via Tseitin promotion) during the preprocessing of ϕ . The inner computation is performed by the standard SPLIT-D method. As Proposition 1 shows, the resulting sum operations will be deterministic, although the formula might not be decision-DNNF.

For both approaches, the knowledge compiler can make use of the following optimization: as the compiler recursively splits and performs unit propagation, stop once the set of reduced clauses contains only projection variables. Run a SAT solver on these clauses and return \top if they are satisfiable and \perp if they are not. This optimized approach is supported by the most recent version of the D4 knowledge compiler [22]. When given a CNF description that also includes declarations of the data variables, it will generate a decision-DNNF representation of the set of data models.

As we will see in the experimental results, terminating the splitting once only projection variables remain allows a projected knowledge compiler to run significantly faster than a standard knowledge compiler on some formulas. It need not generate a detailed representation of the possible assignments to the projection variables. On the other hand, imposing such a strong constraint on the variable selection heuristic for the knowledge compiler can cause it to make suboptimal choices. By allowing splitting on both data and Tseitin variables, SPLIT-DT method lessens the constraints on splitting variable selection. On the other hand, our implementation of SPLIT-DT only allows early termination when the formula has been reduced to the non-Tseitin projection variables. We will see that neither approach consistently outperforms the other.

4.3 The RECOMPILE Method

The RECOMPILE method runs a standard knowledge compiler twice. Given CNF formula ϕ , the first execution, followed by trimming, gives formula $\psi = \text{Trim}(\text{KC}(\phi), P)$ that has the desired set of models, but it potentially contains sum operations that are not deterministic. Formula ψ is converted into a CNF formula with $\phi', Z = \text{Clausify}(\psi)$. Running the compiler a second time and trimming the result gives $\psi' = \text{Trim}(\text{KC}(\phi'), Z)$. This will be a DDNNF representation of $\exists P \phi$.

For the formula illustrated in Figure 1, we can see that the first compilation pass effectively converts the original Plaisted-Greenbaum encoding into a Tseitin encoding, allowing the projection variables to be eliminated via trimming during the second pass. As we will see, recompilation can be very effective, even though it requires two invocations of a knowledge compiler, and the intermediate formula ϕ' will typically be much larger, in terms of clauses, than the original formula ϕ .

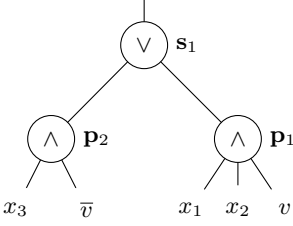
4.4 The REPAIR Method

Our final approach also starts by invoking a standard knowledge compiler to give $\psi = \text{KC}(\phi)$. It recursively traverses ψ to construct DD formula θ . It converts the literals of projection variables into tautologies, but it also examines each sum operation to see if it violates determinism. If so, the compiler inserts correcting terms to remove the overlap between its two arguments.

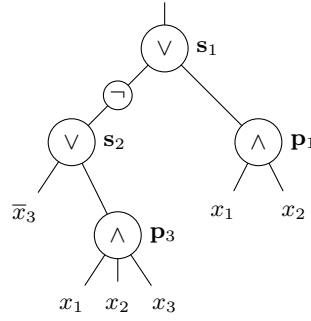
Consider a sum operation $\psi' = \psi_1 \vee \psi_2$ in ψ having associated decision variable x , and assume that the recursion has converted the two arguments into formulas θ_1 and θ_2 . If $x \in D$, then we can return $\theta_1 \vee \theta_2$ as the result of the recursion: x will be its decision variable. If the preprocessor has determined that x is a Tseitin variable, we can also return $\theta_1 \vee \theta_2$ as the result of the recursion; by Proposition 1, the arguments are mutually exclusive.

If the decision variable is a projection variable, we can form a clausal representation ϕ_\wedge of the overlap between the two arguments, i.e., of the formula $\theta_1 \wedge \theta_2$, by creating clausal encodings $\phi_1, Z_1 = \text{Clausify}(\theta_1)$ and $\phi_2, Z_2 = \text{Clausify}(\theta_2)$, and forming a clausal representation of their conjunction as $\phi_\wedge = \phi_1 \cup \phi_2$ with projection variables $Z = Z_1 \cup Z_2$. If ϕ_\wedge unsatisfiable, then the two arguments are mutually exclusive, and therefore the recursion can return $\theta_1 \vee \theta_2$ as the result.

(A) Compiled formula



(B) After REPAIR


Fig. 2. Projection via repair. Term p_3 eliminates the overlap between subformulas x_3 and p_1 .

If the mutual test exclusion fails, we can use the recompilation strategy to generate a DDNNF representation of $\theta_1 \wedge \theta_2$. That is, let $\theta_\wedge = \text{Trim}(\text{KC}(\phi_\wedge), Z)$. As an optimization, we can use this intermediate result to test entailment, either that $\theta_1 \models \theta_2$, or vice-versa, by generating model counts for θ_1 , θ_2 , and θ_\wedge . If the model count for θ_\wedge matches that for θ_1 , then $\theta_1 \models \theta_2$, and therefore we can return θ_2 as the result of the recursion. Similarly, if the model count for θ_\wedge matches that for θ_2 , we can return θ_1 .

Finally, if all these other tests fail, we can introduce a term that excludes the overlap between θ_1 and θ_2 . We would like to generate the result $(\theta_1 \wedge \neg\theta_\wedge) \vee \theta_2$, but the product operation would not be decomposable. Instead, we can use DeMorgan's Laws, generating the formula $\neg(\neg\theta_1 \vee \theta_\wedge) \vee \theta_2$. Both of these sums are deterministic: the first because $\theta_\wedge \models \theta_1$, and therefore $\neg\theta_1$ and θ_\wedge have disjoint models, and the second because we have removed the overlap from the two arguments. The correction term ensures that a computation of the model count only counts the assignments in $\mathcal{M}(\theta_1) \cap \mathcal{M}(\theta_2)$ once. The generated formula will be DD, but with its negations, it will not be in negation-normal form.

Figure 2 illustrates how REPAIR can correct the violation of determinism seen in Figure 1(C). As before, the output of the knowledge compiler (A) split initially on variable v . To repair operation s_1 , the program generates the term $x_1 \wedge x_2 \wedge x_3$ (operation p_3) for the overlap between arguments x_3 and $x_1 \wedge x_2$ (operation p_1 .) DeMorgan's Laws converts the product $x_3 \wedge \neg p_3$ into $\neg(\bar{x}_3 \vee p_3)$.

As we will see with the experimental results, REPAIR can be time consuming, but it can often generate compact results. For example, if all decision variables are either data or Tseitin variables, it will generate the same result as trimming, even when these are not detected during preprocessing. If the knowledge compiler splits only on data variables until only projection variables remain, REPAIR will return the same result as the defer approach but without the optimization of having the compiler stop once only projection variables remain. In addition, having a compiled representation of the overlap between the two arguments to a sum enables an efficient test for entailment, and our experiments have shown that this optimization is often successful. On the other hand, inserting correction terms can cause the generated formula to grow significantly.

5 Implementation and Experimental Evaluation

NOTE TO REVIEWERS: If this paper is accepted, we will submit an artifact including the code (with an MIT license) and raw data from the experiments.

We have implemented a projected knowledge compiler PKC. It invokes D4 by writing a CNF file, running the program, and loading the generated decision-DNNF file. Since many calls to a knowledge compiler may be required when running with the REPAIR method, we also implemented a very lightweight knowledge compiler and use it on formulas with up to 70 clauses. A key optimization is to maintain all of the generated Boolean formulas as a single, directed-acyclic graph (DAG), with each node representing a sum or product operation. New nodes are added only after simplifications have been performed, and when no prior node with the same operation and arguments exists. A formula can be extracted from the DAG by recursive traversal from its root node.

Following compilation, PKC generates a count of the number of models. If the input CNF file specifies weights for the data variables, it also computes a weighted model count. We implemented our own package, based on one described by Bryant, et al. [2], for performing arithmetic with numbers of the form $a \cdot 2^b \cdot 5^c$, where a is implemented as an unbounded integer, and both b and c are represented as 32-bit, signed integers. This enables scaling by arbitrary powers of two, as is required for performing model counting via weighted counting. It can also perform exact decimal arithmetic, handling the weights used in the weighted model counting competitions. Performing exact arithmetic is essential when computing weighted counts for formulas with negation. For example, a bounded-precision representation can lose all accuracy when computing $(1 - (1 - w))$ for values of w close to 0. In our measurements, the time required to compute these counts is insignificant compared to the time for the compilation.

5.1 Majority Formulas

As a benchmark representing formulas that require auxiliary variables to have polynomially sized CNF representations, we generated formulas encoding Maj_n . Given a set of input variables $X = \{x_1, x_2, \dots, x_n\}$, this formula evaluates to true when over $n/2$ of the inputs are assigned value 1. Our CNF representation uses auxiliary variables of the form $m_{i,j}$ that evaluate to 1 when at least j of the variables in $\{x_1, x_2, \dots, x_i\}$ are assigned true [29]. The defining clauses for $m_{i,j}$ encode the recurrence $m_{i,j} = m_{i-1,j} \vee (x_i \wedge m_{i-1,j-1})$. The majority formula is then $m_{n,k}$, where $k = \lfloor n/2 \rfloor + 1$. Each auxiliary variable requires four defining clauses for a Tseitin encoding and two for Plaisted-Greenbaum.

Figure 3 show the elapsed times required to run PKC on majority formulas. For the Tseitin encodings (A), we can simply run the knowledge compiler and trim the result. This can readily handle the formula for $n = 99$ in under 15 seconds. Running REPAIR yields the same projected formula, since each mutual exclusion test is successful. We show two different plots: without (REPAIR-A) and with (REPAIR-B) using the preprocessor to detect Tseitin variables. Without preprocessing, the time required by the SAT solver to detect mutual exclusion becomes significant. With preprocessing, which runs very fast, the times match those for trimming. The other methods reached our time limit

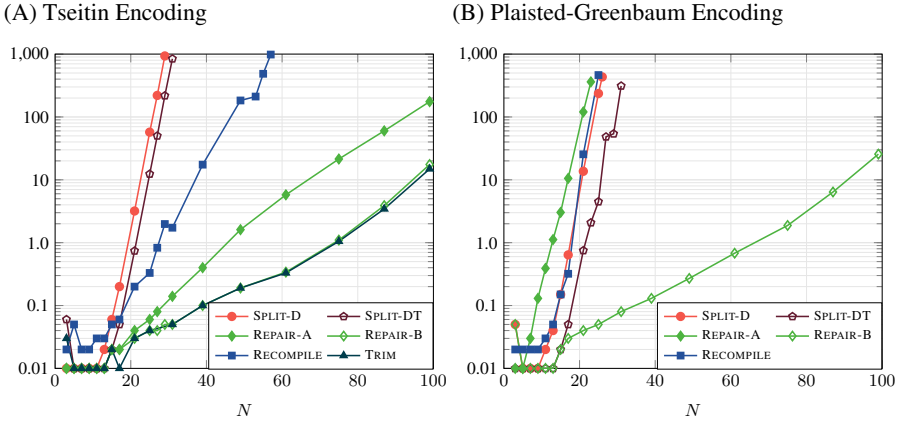


Fig. 3. Runtimes (in seconds) for projected compilation of Maj_n formulas with two different encodings of Boolean operations.

of 1000 seconds for smaller values of n : 57 for RECOMPILE, 31 for SPLIT-DT, and 29 for SPLIT-D.

The poor performance for SPLIT-DT in Figure 3(A) indicates a major shortcoming when using D4 to perform projected knowledge compilation. For these formulas, all of the projection variables are identified as Tseitin variables, and so, in principle, D4 should be able to achieve the same performance running SPLIT-DT as we saw with trimming and REPAIR-B. Unfortunately, when D4 is invoked in projected mode, it disables the use of hypergraph partitioning for detecting opportunities to split a subformula into ones having disjoint sets of variables [21], a key source of D4’s performance.

No method works well on unmodified formulas with the Plaisted-Greenbaum encoding (B). All hit the 1000-second time limit when n exceeds 26. Standard knowledge compilation of these formulas benefits greatly from hypergraph partitioning, but running SPLIT-D causes D4 to disable this feature. Having to generate correcting terms for many of the sum operations adds complications to REPAIR-A. Unlike our simple example, the initial compilation fails to extract the structure of the underlying formulas for RECOMPILE. On the other hand, our preprocessor is able to promote all projection variables into Tseitin variables, enabling both SPLIT-DT and REPAIR-B to match their performance for the Tseitin encoding in (A).

5.2 Projected Model Counting Competition Benchmark Formulas

Model checking competitions were held in 2022 and 2023 with tracks for projected model counting and for projected, weighted model counting.³ For each track, 100 benchmark formulas were publicly available in advance, while 100 are kept private until the competition. We downloaded the 400 public formulas from the two tracks for both

³ https://mcccompetition.org/2022/mc_description.html and https://mcccompetition.org/2023/mc_description.html

years. Of these we found that 371 were unique, where two formulas were deemed identical if both their sets of clauses and their sets of data variables were identical.

We performed knowledge compilation for the 377 formulas with the latest version of the D4, setting a time limit of 2000 seconds. This completed for 124 formulas, and these became our benchmark set. We ran PKC on these formulas using all methods except TRIM. For each of these, we first performed unit propagation and pure literal elimination as described in Section 4. For SPLIT-DT and REPAIR, we ran the preprocessor to detect and promote Tseitin variables. The runtimes include preprocessing, compilation, and (weighted) counting. We allowed total runtimes up to 4000 seconds.

Figure 4 compares the performance of PKC (Y-axis) with that of D4 performing standard knowledge compilation (X-axis), where (A) compares the runtimes, while (B) compares the sizes of the compiled results. Size is measured as the number of clauses in a CNF representation of the compiled result. An operation with k arguments requires $k + 1$ clauses, and so the size also equals the sum of the number of operations plus the total number of arguments in a DAG representation of the formula. Each vertical line in the figure connects compilations of a single formula by different methods, with those going to the top of plot (A) indicating that at least one of the methods timed out.

Looking first at the time performance (A), we can see several important results:

- The tall, vertical lines connecting different compilations of single formulas demonstrate a very large variance in the runtimes of the different methods. Some formulas requiring thousands of seconds in one mode could be compiled in fractions of a second by another.
- Projected knowledge compilation generally requires more time than standard knowledge compilation, as indicated by the majority of the points being above the diagonal. As an extreme case, the single formula for which all methods timed out required only 3.3 seconds for standard knowledge compilation. On the other hand, projected knowledge compilation outperformed standard knowledge compilation for a number of cases, including ones where standard compilation required hundreds of seconds, while projected knowledge compilation required only fractions of a second. When running SPLIT-D or SPLIT-DT, D4 can take advantage of early termination. In addition, pure literal elimination greatly simplified some of the formulas.
- Overall, the SPLIT-D method achieved the best results. It generally ran faster than the others, and it completed the most (117) formulas. However, there were some formulas for which one of the other methods outperformed SPLIT-D, sometimes by orders of magnitude.

The results for the formula sizes (B) are also quite revealing:

- Although projected compilation could yield results that are exponentially larger than standard knowledge compilation, only 25 of the 372 data points show results where the number of clauses in the projected compilation exceeds those in the standard compilation. Indeed, many of the projected formulas are much smaller than those without projection, including some by orders of magnitude.
- Compared to the wide range in runtimes for the different compilation methods, their generated results were closer in size, and the SPLIT-D method is less dominant.

Table 2. Performance of different modes of PKC on 124 benchmark problems. Scores are based on which methods performed best, with credit for ties divided evenly.

Method	Completed	Runtime		Formula Size	
		Score	Par-2 Avg.	Score	Par-2 Avg.
SPLIT-D	117	49.6	650.3	46.2	126,916,226
SPLIT-DT	113	49.6	921.2	33.7	192,828,126
RECOMPILE	72	14.9	3575.9	16.5	838,811,564
REPAIR	70	8.9	3649.9	16.7	871,279,244
Virtual Best	123	123.0	247.3	123.0	31,027,774

Table 2 summarizes the relative performance of the different methods for the 124 benchmarks. It includes a “Virtual Best” method that for each formula, achieves both the minimum runtime, as well as the minimum formula size of the four methods evaluated. The entries labeled “Score” were computed by dividing one point for each benchmark evenly among those methods that tied for best performance. We count all methods running within 5% of the best as having tied. For runtimes, we also counted all times below 1.0 second as ties. The Par-2 averages were computed by charging an execution that timed out with 8000 seconds for runtime and 1.5×10^9 clauses for formula size (the largest formula generated had around 750 million clauses.)

As can be seen, the SPLIT-D method achieves the best overall performance, including completing within 4000 seconds for 117 of the 123 formulas that completed for any mode. Allowing splitting on Tseitin variables (SPLIT-DT) did not perform as well overall, indicating that the freedom to split on more variables does not compensate for the inability of our implementation to terminate until only non-Tseitin projection variables remain. On the other hand, exploiting Tseitin variables enables SPLIT-DT to complete for four benchmarks that timed out with SPLIT-D. Both RECOMPILE and REPAIR have poor Par-2 averages, largely because of their low number of completions. Nonetheless, each method had its formulas for which it outperformed the others.

The much better Par-2 performance of Virtual Best suggests finding ways to combine the methods. Indeed, the program could hypothetically achieve a Par-2 performance of 568.7 and complete 119 benchmarks by the following simple “portfolio” approach: run REPAIR for a maximum of 90 seconds. If that computation fails to complete, then start over with SPLIT-D. There is some logic behind such an approach: running REPAIR allows D4 to use hypergraph partitioning. For some benchmarks, D4 can then run much faster and generate much smaller formulas, even if some amount of repairing is required for the sum operations.

6 Conclusion

Being able to remove auxiliary variables is a key capability for a knowledge compiler. In addition to the projected compilation method already implemented in D4, we have presented further refinements and alternative approaches. The wide range of timings for the different methods suggests that hybrids and further refinements could improve compiler stability and overall performance.

References

1. Aziz, R.A., Chu, G., Muise, C., Stuckey, P.: # \exists SAT: Projected model counting. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 9340, pp. 121–137 (2015)
2. Bryant, R.E., Nawrocki, W., Avigad, J., Heule, M.J.H.: Certified knowledge compilation with application to verified model counting. In: Theory and Applications of Satisfiability Testing (SAT). Schloss Dagstuhl (July 2023)
3. Burch, J.R., Clarke, E.M., Long, D.E., McMillan, K.L., Dill, D.L.: Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **13**(4), 401–424 (1994)
4. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. *Information and Computation* **98**(2), 142–170 (1992)
5. Chavira, M., Darwiche, A.: On probabilistic inference by weighted model counting. *Artificial Intelligence* **172**, 772–799 (2008)
6. Darwiche, A.: Decomposable negation normal form. *Journal of the ACM* **48**(4), 608–647 (2001)
7. Darwiche, A.: On the tractable counting of theory models and its application to truth maintenance and belief revision. *Journal of Applications of Non Classical Logics* **11**(1-2), 11–34 (2001)
8. Darwiche, A.: New advances in compiling CNF to decomposable negation normal form. In: European Conference on Artificial Intelligence. pp. 328–332 (2004)
9. Darwiche, A., Marquis, P.: A knowledge compilation map. *Journal of Artificial Intelligence Research* **17** (2002)
10. Dudek, J.M., Phan, V.H.N., Vardi, M.Y.: ADDMC: Weighted model counting with algebraic decision diagrams. In: AAAI Conference on Artificial Intelligence. pp. 1468–1475 (2020)
11. Dudek, J.M., Phan, V.H.N., Vardi, M.Y.: ProCount: Weighted projected model counting with graded project-join trees. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 12831, pp. 152–170 (2021)
12. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 3569, pp. 61–75 (2005)
13. Fleury, M., Biere, A.: Mining definitions in Kissat with Kittens. *Formal Methods in System Design* **60**(3), 381–404 (2022)
14. Gomes, C.P., Sabharwal, A., Selman, B.: Model counting. In: Handbook of Satisfiability, pp. 633–654. IOS Press (2009)
15. Iser, M., Manthey, N., Sinz, C.: Recognition of nested gates in CNF formulas. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 9340, pp. 255–271 (2015)
16. Järvisalo, M., Biere, A., Heule, M.: Blocked clause elimination. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 6015, pp. 129–144 (2010)
17. Järvisalo, M., Biere, A., Heule, M.J.H.: Simulating circuit-level simplifications on CNF. *Journal of Automated Reasoning* **49**(4), 583–619 (2012)
18. Kimmig, A., den Broeck, G.V., Raedt, L.D.: Algebraic model counting. *Journal of Applied Logic* **22**, 46–62 (July 2017)
19. Lagniez, J.M.: Knowledge compilation using theory prime implicates. In: International Joint Conference on Artificial Intelligence (IJCAI). pp. 837–843 (1995)
20. Lagniez, J.M., Lonca, E., Marquis, P.: Definability for model counting. *Artificial Intelligence* **281**, 103229 (April 2020)
21. Lagniez, J.M., Marquis, P.: An improved decision-DNNF compiler. In: International Joint Conference on Artificial Intelligence (IJCAI). pp. 667–673 (2017)

22. Lagniez, J.M., Marquis, P.: A recursive algorithm for projected model counting. In: AAAI Conference on Artificial Intelligence. pp. 1536–1543 (2019)
23. Monet, M., Olteanu, D.: Towards deterministic decomposable circuits for safe queries. In: Alberto Mendelzon International Workshop on Foundations of Data Management (AMW) (2018)
24. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Design Automation Conference (DAC). pp. 530–535. ACM/IEEE (2001)
25. Muise, C., McIlraith, S.A., Beck, J.C., Hsu, E.: DSHARP: Fast d-DNNF compilation with sharpSAT. In: Canadian Conference on Artificial Intelligence. LNCS, vol. 7310, pp. 356–361 (2012)
26. Oztok, U., Darwiche, A.: On compiling CNF into decision-DNNF. In: Constraint Programming (CP). LNCS, vol. 8656, pp. 42–57 (2014)
27. Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. *Journal of Symbolic Computation* 2(3), 293–304 (1986)
28. Prestwich, S.: CNF encodings. In: *Handbook of Satisfiability*, pp. 75–97. IOS Press (2009)
29. Sinz, C.: Towards an optimal CNF encoding of Boolean cardinality constraints. In: *Principles and Practice of Constraint Programming (CP)*. LNCS, vol. 3709, pp. 827–831 (2005)

A Proofs of Key Results

This appendix contains details that were omitted from the main paper due to space limitations.

A.1 Tseitin Variables

As background, we prove the converse of Proposition 1

Proposition 4. *If projection variable z partitions the set of data models for Boolean formula ψ into disjoint subsets, i.e., $\mathcal{M}_{\mathcal{D}}^{\bar{z}}(\psi) \cap \mathcal{M}_{\mathcal{D}}^z(\psi) = \emptyset$, then z is a Tseitin variable*

Any assignment $\alpha \in \mathcal{M}_{\mathcal{D}}(\psi)$ must be in either $\mathcal{M}_{\mathcal{D}}^{\bar{z}}(\psi)$ or $\mathcal{M}_{\mathcal{D}}^z(\psi)$, but by the premise, it is not in both. In the former case, ψ will only be satisfied by assignments $\alpha \cdot \beta$ such that $\beta(z) = 0$ and in the latter by ones such that $\beta(z) = 1$. Therefore, there cannot be assignments $\beta_1, \beta_2 \in \mathcal{U}(P)$ such that both $\alpha \cdot \beta_1$ and $\alpha \cdot \beta_2$ are in $\mathcal{M}(\psi)$, but $\beta_1(z) \neq \beta_2(z)$. \square

A.2 Tseitin Variable Promotion

Given formula ϕ and projection variable z , our method for promoting z to be a Tseitin variable is based on the following conditions. Let ℓ be either z or \bar{z} , and let ϕ_ℓ denote those clauses in ϕ containing ℓ . Assume ϕ_ℓ consists of k clauses: C_1, C_2, \dots, C_k . We require the following conditions:

1. Dependency set $\mathcal{V}(\phi_\ell)$ contains only data variables, previously detected Tseitin variables, and z .
2. There are no complementary literals in ϕ_ℓ , i.e., with both literals x and \bar{x} occurring for some variable x .

Promotion involves adding all blocked clauses B of the form $B = [\bar{\ell} \vee \bar{\ell}_1 \vee \bar{\ell}_2 \vee \dots \vee \bar{\ell}_k]$, where each $\ell_i \in C_i$ and $\ell_i \neq \ell$ for $1 \leq i \leq k$. The second condition guarantees that B will not contain complementary literals, and so it will not be a tautology.

Proposition 5. *If assignment ρ satisfies ϕ , i.e., $\rho \in \mathcal{M}(\phi)$, but it does not satisfy clause B , then we can construct another assignment ρ' that satisfies both ϕ and B .*

As proof, we can see that to falsify B , assignment ρ must include ℓ , as well as literals ℓ_i for $1 \leq i \leq k$. Let $\rho' = (\rho - \{\ell\}) \cup \{\bar{\ell}\}$. That is, we obtain ρ' from ρ by negating literal ℓ . This assignment now satisfies B . The only clauses in ϕ that could become falsified by this change are those in ϕ_ℓ , but for each i , literal ℓ_i will satisfy clause C_i , and so ρ' continues to satisfy ϕ . \square

Let ϕ' consist of the clauses of ϕ plus all of the added blocked clauses.

Proposition 6. *Projection variable z is a Tseitin variable for formula ϕ' .*

Consider data assignment $\alpha \in \mathcal{M}_{\mathcal{D}}^{\bar{\ell}}(\phi')$. The first condition guarantees that α induces a unique assignment for all projection variables occurring in ϕ_{ℓ} other than z , defining a unique partial assignment β' for these variables. In addition, $\alpha \cdot \beta'$ must satisfy each clause C_i and therefore, since ℓ is falsified, there must be some literal ℓ_i such that both $\ell_i \in \alpha \cdot \beta'$ and $\ell_i \in C_i$. On the other hand, any total assignment $\alpha \cdot \beta$ such that β is consistent with β' will satisfy the corresponding blocked clause B (B is not a tautology) only when ℓ is falsified, implying that $\alpha \notin \mathcal{M}_{\mathcal{D}}^{\ell}(\phi')$. By Proposition 4, z is therefore a Tseitin variable. \square