Towards Practical First-Order Model Counting

Ananth K. Kidambi[1*], **Guramrit Singh**[1], **Paulius Dilkas**[2], **Kuldeep S. Meel**[3]

[1]Indian Institute of Technology Bombay, Mumbai, India
[2]National University of Singapore, Singapore, Singapore
[3]University of Toronto, Toronto, Canada
{210051002, 210050061}@iitb.ac.in, paulius.dilkas@nus.edu.sg, meel@cs.toronto.edu

Abstract

First-order model counting (FOMC) is the problem of counting the number of models of a sentence in first-order logic. Recently, a new algorithm for FOMC was proposed that, instead of simply providing the final count, generates definitions of (possibly recursive) functions, which can be evaluated with different arguments to compute the model count for any domain size. However, the algorithm did not include base cases in the recursive definitions. This work makes three contributions. First, we demonstrate how to construct function definitions that include base cases by modifying the logical formulas used in the FOMC algorithm. Second, we extend the well-known circuit modification technique in knowledge compilation, known as smoothing, to work with the formulas corresponding to base cases and the recently proposed node types, such as domain recursion and constraint removal. Third, we introduce a compilation algorithm that transforms the function definitions into C++ code, equipped with arbitrary-precision arithmetic. These additions allow the new FOMC algorithm to scale to domain sizes over 9000 times larger than the current state of the art, as demonstrated through experimental results.

1 Introduction

First-order model counting is the problem of counting the number of models of a sentence in first-order logic over some given domain(s). The (symmetric) weighted variant of the problem (WFOMC) asks for the sum of model weights instead, where the weight of a model is a product over predicate weights (Van den Broeck et al. 2011). WFOMC is a prominent approach to the broader problem known as *lifted (probabilistic) inference* that aims to compute probabilities more efficiently by utlilising symmetries inherent in the description of the problem (Kersting 2012).

Lifted inference is an active area of research with recent work in domains such as constraint satisfaction problems (Totis et al. 2023) and probabilistic answer set programming (Azzolini and Riguzzi 2023). WFOMC itself has been used to perform inference on probabilistic databases (Gribkoff, Suciu, and Van den Broeck 2014) and probabilistic logic programs (Riguzzi et al. 2017). By iteratively considering domains of increasing sizes, the model count of a formula can be interpreted as an (integer) sequence. WFOMC algorithms have been used to discover new sequences (Svatos et al. 2023) and conjecture (Barvínek et al. 2021) and construct (Dilkas and Belle 2023) recurrence relations and other recursive structures that describe these sequences. Recently, WFOMC algorithms have also been extended and applied to perform *sampling* (Wang et al. 2022; Wang et al. 2023).

The complexity of WFOMC is typically characterised in terms of *data complexity*, i.e., fixing the formula and asking whether there exists an algorithm that can compute the WFOMC in time polynomial in the domain size(s). If such an algorithm exists, the formula is called *liftable* (Jaeger and Van den Broeck 2012). Beame et al. (2015) showed that there exists an unliftable formula with three variables. It is also known that all formulas with up to two variables are liftable (Van den Broeck 2011; Van den Broeck, Meert, and Darwiche 2014). This liftable fragment of formulas with two variables has been extended with various axioms (Tóth and Kuželka 2023; van Bremen and Kuželka 2023), counting quantifiers (Kuželka 2021), and in other ways (Kazemi et al. 2016).

There is a variety of WFOMC algorithms with different underlying principles. Perhaps the most prominent class of WFOMC algorithms is based on *first-order knowledge compilation*. Here, by iteratively applying *compilation rules*, the formula is compiled to a representation (such as a circuit of graph), which can then be used to compute the WFOMC for any combination of domain sizes (or weights). Algorithms in this class include FORCLIFT (Van den Broeck et al. 2011) and its recent extension CRANE (Dilkas and Belle 2023). The former compiles formulas into circuits whereas the latter compiles them first to *first-order computational graphs* (FCGs) and then to (algebraic) equations. Another major WFOMC algorithm FASTWFOMC (van Bremen and Kuželka 2021) is based on cell enumeration. Other algorithms employ local search (Niu et al. 2011), junction trees (Venugopal, Sarkhel, and Gogate 2015), Monte Carlo sampling (Gogate and Domingos 2016), and anytime approximation via upper/lower bound construction (van Bremen and Kuželka 2020).

The recently-proposed CRANE algorithm, while able to handle formulas beyond the reach of FORCLIFT, was incomplete in that it could only construct functions defini-

*The first and second authors contributed equally and were affiliated with the National University of Singapore during the completion of this work.

tions, which would then need to be evaluated to compute the WFOMC. Moreover, recursive functions were presented without base cases. In this work, we present CRANE2—an extension of CRANE that fixes these two weaknesses.

Figure 1 outlines the workflow of the new algorithm. In Section 3, we describe how `CompileWithBaseCases` finds the base cases for recursive functions by: (i) identifying a sufficient set of base cases for each function, (ii) constructing formulas that correspond to these base cases, and (iii) recursing on these new formulas. Then, Section 4 describes post-processing techniques for FCGs and the formulas from Step (ii) above necessary to preserve the correct model count. Next, Section 5 explains how function definitions that encode a solution to a WFOMC problem are then compiled to C++ programs.[1] Finally, Section 6 reports on two experiments that compare CRANE2 with other state-of-the-art WFOMC algorithms.

2 Preliminaries

In Section 2.1, we provide a summary of the basic principles of first-order logic. Then, in Section 2.2, we formally define WFOMC and discuss the distinctions between three variations of first-order logic that are utilised for WFOMC. Finally, in Section 2.3, we introduce the terminology used to describe the output of the original CRANE algorithm, i.e., functions and equations that define them.

We use $\mathbb{N}_0$ to represent the set of non-negative integers. In both algebra and logic, we write $S\sigma$ to denote the application of a *substitution* σ to an expression S, where $\sigma = [x_1 \mapsto y_1, x_2 \mapsto y_2, \ldots, x_n \mapsto y_n]$ signifies the replacement of all instances of x_i with y_i for all $i = 1, \ldots, n$.

2.1 First-Order Logic

In this section, we will review the basic concepts of first-order logic as they are used in first-order knowledge compilation algorithms. There are two key differences between the logic used within such algorithms and the logic supported as input. First, Skolemization (Van den Broeck, Meert, and Darwiche 2014) is employed to eliminate existential quantifiers by introducing additional predicates. Second, the input formula is rewritten as a conjunction of clauses, each of which is in *prenex normal form* (Hinman 2018).

A *formula* is a conjunction of clauses. A *clause* is of the form $\forall x_1 \in \Delta_1 . \forall x_2 \in \Delta_2 \ldots \forall x_n \in \Delta_n . \phi(x_1, x_2, \ldots, x_n)$, where ϕ is a disjunction of literals that only contain variables $x_1, \ldots, x_n$ (and any constants). We say that a clause is a *(positive) unit clause* if (i) there is only one literal with a predicate, and (ii) it is a positive literal. A *literal* is either an atom (i.e., a *positive* literal) or its negation (i.e., a *negative* literal). An *atom* is either (i) $P(t_1, \ldots, t_m)$ for some predicate P and terms $t_1, \ldots, t_m$ (written as $P(\mathbf{t})$ for short) or (ii) $x = y$ for some terms x and y. An atom is *ground* if it contains no variables, i.e., only constants. The *arity* of a predicate is the number of arguments it takes, i.e., m in the case of predicate P mentioned above. When we want to denote a predicate together with its arity, we write P/m. A *term* is either a variable or a constant. Throughout the paper, we will use set-theoretic notation, interpreting a formula as a set of clauses and a clause as a set of literals. Moreover, for readability, clauses written on separate lines are implicitly conjoined.

2.2 WFOMC Algorithms and Their Logics

In Table 1, we outline the differences among three first-order logics commonly used in WFOMC: (i) FO is the input format for FORCLIFT[2] and its extensions CRANE[3] and CRANE2; (ii) C^2 is often used in the literature on FASTWFOMC and related methods (Kuželka 2021; Malhotra and Serafini 2022); (iii) $\mathsf{UFO}^2 + \mathsf{EQ}$ is the input format supported by a private version of FASTWFOMC obtained directly from the authors. Note that the publicly available version[4] of FASTWFOMC does not support any cardinality constraints. The notation we use to refer to each logic is standard in the case of C^2, new in the case of $\mathsf{UFO}^2 + \mathsf{EQ}$, and redefined to be more specific in the case of FO. All three logics are function-free, and domains are always assumed to be finite.

In FO, each term is assigned to a *sort*, and each predicate P/n is assigned to a sequence of n sorts. Each sort has its corresponding domain. Most of these assignments to sorts are typically left implicit and can be reconstructed from the quantifiers. For example, $\forall x, y \in \Delta .\ P(x, y)$ implies that variables x and y have the same sort. On the other hand, $\forall x \in \Delta .\ \forall y \in \Gamma .\ P(x, y)$ implies that x and y have different sorts, and it would be improper to write, for example, $\forall x \in \Delta .\ \forall y \in \Gamma .\ P(x, y) \vee x = y$. FO is also the only logic to support constants, formulas with more than two variables, and the equality predicate.

Remark. In the case of FORCLIFT and its extensions, support for a formula as valid input does not imply that the algorithm can compile the formula into a circuit or graph suitable for lifted model counting. However, it is known that FORCLIFT compilation is guaranteed to succeed on any FO formula without constants and with at most two variables (Van den Broeck 2011; Van den Broeck, Meert, and Darwiche 2014).

Both C^2 and $\mathsf{UFO}^2 + \mathsf{EQ}$ are single-sorted (i.e., variables are always quantified over the same domain) and only include formulas with at most two variables and no constants. Unlike $\mathsf{UFO}^2 + \mathsf{EQ}$, C^2 supports existential quantifiers, including *counting quantifiers*. For example, $\exists^{=1} x .\ \phi(x)$ means that there exists *exactly one* x such that $\phi(x)$, and $\exists^{\leq 2} x .\ \phi(x)$ means that there exist *at most two* such x. The advantage $\mathsf{UFO}^2 + \mathsf{EQ}$ brings over C^2 is the support for *(equality) cardinality constraints*. For example, $|P| = 3$ constrains all models to have *exactly three positive literals with predicate* P.

Definition 1 (Model)**.** *Let ϕ be a formula in* FO*. For each predicate P/n in ϕ, let $(\Delta_i^P)_{i=1}^n$ be a list of the corresponding domains (which may not be distinct). Let σ be a map from*

[1] Compilation to C++ has been explored before (Kazemi and Poole 2016), although with limited capabilities in terms of what formulas can be handled in a lifted manner.

[2] https://github.com/UCLA-StarAI/Forclift

[3] https://doi.org/10.5281/zenodo.8004077

[4] https://comp.nus.edu.sg/~tvanbr/software/fastwfomc.tar.gz

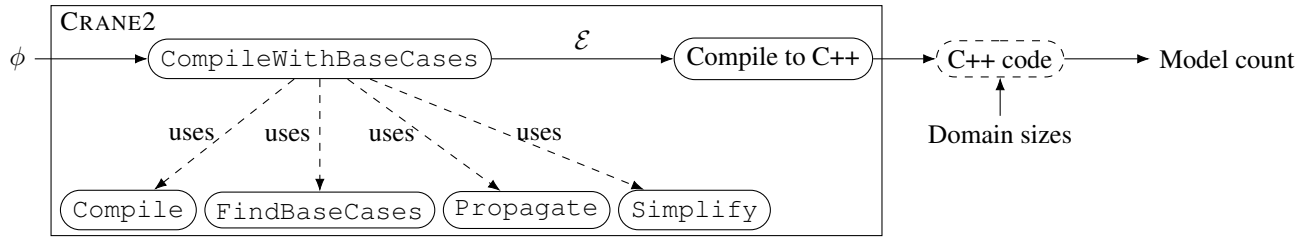

Figure 1: The outline of using CRANE2 to compute the model count of a formula ϕ. First, the formula is compiled into a set of equations, which are then used to create a C++ program. This program can be executed with different command line arguments to calculate the model count of ϕ for different domain sizes. To accomplish this, the `CompileWithBaseCases` function employs several components: (i) the knowledge compilation algorithm of CRANE, referred to as `Compile`, (ii) a procedure called `FindBaseCases`, which identifies a sufficient set of base cases, (iii) a procedure called `Propagate`, which constructs a formula corresponding to a given base case, and (iv) algebraic simplification techniques (denoted as `Simplify`).

Logic	Sorts	Constants	Variables	Quantifiers	Additional atoms
FO	one or more	✓	unlimited	$\forall$, $\exists$	$x = y$
C^2	one	✗	two	$\forall$, $\exists$, $\exists^{=k}$, $\exists^{\leq k}$, $\exists^{\geq k}$	—
$\mathsf{UFO}^2 + \mathsf{EQ}$	one	✗	two	$\forall$	$\lvert P\rvert = m$

Table 1: Comparison of three logics used in WFOMC based on the following aspects: (i) number of sorts, (ii) support for constants, (iii) maximum number of variables, (iv) supported quantifiers, and (v) supported atoms in addition to those of the form $P(\mathbf{t})$ for a predicate P/n and n-tuple of terms $\mathbf{t}$. In this context: (i) k and m are non-negative integers, with the latter depending on the domain size, (ii) P represents a predicate, and (iii) x and y are terms.

the domains of ϕ to their interpretations as sets that, satisfying the following conditions: (i) the sets are pairwise disjoint, and (ii) the constants in ϕ are included in the corresponding domains. Then a structure *of ϕ (with respect to σ) is a set M of ground literals defined by adding either $P(\mathbf{t})$ or $\neg P(\mathbf{t})$ for every predicate P/n in ϕ and n-tuple $\mathbf{t} \in \prod_{i=1}^{n} \sigma(\Delta_i^P)$. A structure is considered a* model *if it satisfies ϕ.*

Definition 2 (WFOMC (Van den Broeck et al. 2011))**.** *Continuing from Definition 1, for each predicate P in ϕ, let $w^+(P)$ and $w^-(P)$ be its* weights *in $\mathbb{Q}$. Unless specified otherwise, we assume weights to be equal to one. The* (symmetric) weighted first-order model count *(WFOMC) of ϕ (with respect to σ, w^+, and w^-) is calculated as follows:*

$$\sum_{M \models \phi} \prod_{P(\mathbf{t}) \in M} w^+(P) \prod_{\neg P(\mathbf{t}) \in M} w^-(P),$$

where the sum is taken over all models of ϕ.

Example 1 (Counting functions)**.** *To define predicate P as an endofunction on Δ, in C^2 one would write $\forall x \in \Delta.\ \exists^{=1} y \in \Delta.\ P(x, y)$. In $\mathsf{UFO}^2 + \mathsf{EQ}$, the same could be written as*

$$\begin{gathered} \forall x, y \in \Delta.\ S(x) \vee \neg P(x, y) \\ |P| = |\Delta|, \end{gathered} \tag{1}$$

where $w^-(S) = -1$. Although Formula (1) *has more models compared to its counterpart in C^2, the weight function makes the* weighted *model count of Formula* (1) *equal to the number of endofunctions on Δ.*

Equivalently, in FO we would write

$$\begin{gathered} \forall x \in \Gamma.\ \exists y \in \Delta.\ P(x, y) \\ \forall x \in \Gamma.\ \forall y, z \in \Delta.\ P(x, y) \wedge P(x, z) \Rightarrow y = z. \end{gathered} \tag{2}$$

The first clause asserts that each x must have at least one corresponding y, while the second statement adds the condition that if x is mapped to both y and z, then y must equal z. It is important to note that Formula (2) *is written with two domains instead of just one. However, we can still determine the number of endofunctions by assuming that the sizes of Γ and Δ are equal. This formulation, as observed by Dilkas and Belle (2023), can prove beneficial in enabling first-order knowledge compilation algorithms to find efficient solutions.*

2.3 Algebra

We write `expr` to represent an arbitrary algebraic expression. It is important to note that certain terms have different meanings in the context of algebra compared to logic. In algebra, a *constant* refers to a non-negative integer. Likewise, a *variable* can either be a parameter of a function or a variable introduced through summation, such as i in the expression $\sum_{i=1}^{n} \texttt{expr}$. A (function) *signature* is denoted as $f(x_1, \ldots, x_n)$ (or $f(\mathbf{x})$ for short), where f represents an n-ary function, and each x_i represents a variable. An *equation* is written as $f(\mathbf{x}) = \texttt{expr}$, with $f(\mathbf{x})$ representing a signature.

Definition 3 (Base case)**.** *Let $f(\mathbf{x})$ be a function call where each x_i is either a constant or a variable (note that signatures are included in this definition). Then a function call $f(\mathbf{y})$ is considered a* base case *of $f(\mathbf{x})$ if $f(\mathbf{y}) = f(\mathbf{x})\sigma$, where σ is a substitution that replaces one or more x_i with a constant.*

Algorithm 1: `CompileWithBaseCases(`ϕ`)`

Input: formula ϕ
Output: set $\mathcal{E}$ of equations

```
1 (E, F, D) ← Compile(φ);
2 E ← Simplify(E);
3 foreach base case f(x) ∈ FindBaseCases(E) do
4     ψ ← F(f);
5     foreach i such that x_i ∈ ℕ_0 do
6         ψ ← Propagate(ψ, D(f, i), x_i);
7     E ← E ∪ CompileWithBaseCases(ψ);
```

Algorithm 2: `FindBaseCases(`$\mathcal{E}$`)`

Input: set $\mathcal{E}$ of equations
Output: set $\mathcal{B}$ of base cases

```
1 B ← ∅;
2 foreach equation (f(x) = expr) ∈ E do
3     foreach function call f(y) ∈ expr do
4         foreach y_i ∈ y do
5             if y_i ∈ ℕ_0 then
6                 B ← B ∪ { f(x)[x_i ↦ y_i] };
7             else if y_i = x_i − c_i for some c_i ∈ ℕ_0 then
8                 for j ← 0 to c_i − 1 do
9                     B ← B ∪ { f(x)[x_i ↦ j] };
```

3 Completing the Definitions of Recursive Functions

Algorithm 1 outlines our overall approach for compiling a formula into a set of equation that include the required base cases. In short, we first use the knowledge compilation algorithm of the original CRANE to compile the formula into: (i) set $\mathcal{E}$ of equations, (ii) map $\mathcal{F}$ from function names to formulas, and (iii) map $\mathcal{D}$ from function names and argument indices to domains. After some algebraic simplification, $\mathcal{E}$ is passed to the `FindBaseCases` procedure (described in Section 3.1) that returns a set of base cases that we need to find solutions for. For each base case $f(\mathbf{x})$, we identify the formula associated with f and simplify it using the `Propagate` procedure (described in Section 3.2). The algorithm then recurses on these simplified formulas and adds the resulting base case equations to $\mathcal{E}$. Example 2 explains Algorithm 1 in more detail.

Example 2 (Counting bijections)**.** *Consider the following formula (previously examined by Dilkas and Belle (2023)) that defines predicate P to be a bijection between two sets Γ and Δ:*

$$\forall x \in \Gamma.\ \exists y \in \Delta.\ P(x, y)$$
$$\forall y \in \Delta.\ \exists x \in \Gamma.\ P(x, y)$$
$$\forall x \in \Gamma.\ \forall y, z \in \Delta.\ P(x, y) \wedge P(x, z) \Rightarrow y = z$$
$$\forall x, z \in \Gamma.\ \forall y \in \Delta.\ P(x, y) \wedge P(z, y) \Rightarrow x = z.$$

In particular, we examine the first solution that CRANE2 *returns for this formula.*

After lines 1 and 2, we have

$$\mathcal{E} = \left\{ \begin{array}{l} f(m,n) = \sum_{l=0}^{n} \binom{n}{l} (-1)^{n-l} g(l,m), \\ \\ g(l,m) = g(l-1,m) + m g(l-1,m-1) \end{array} \right\};$$

$$\mathcal{D} = \{\, (f,1) \mapsto \Gamma, (f,2) \mapsto \Delta, (g,1) \mapsto \Delta^{\top}, (g,2) \mapsto \Gamma \,\},$$

where $\Delta^{\top}$ is a new domain introduced by `Compile`*. Then* `FindBaseCases` *identifies two base cases: $g(0, m)$ and $g(l, 0)$. In both cases,* `CompileWithBaseCases` *recurses on the formula $\mathcal{F}(g)$ simplified by assuming that one of the domains is empty. In the first case, we recurse on the formula $\forall x \in \Gamma.\ S(x) \vee \neg S(x)$, where S is a predicate introduced by Skolemization with weights $w^{+}(S) = 1$ and $w^{-}(S) = -1$. Hence, we get the base case $g(0, m) = 0^m$. In the case of $g(l, 0)$,* `Propagate(`ψ`,` Γ`,` 0`)` *returns an empty formula, giving us $g(l, 0) = 1$.*

Note that these base cases overlap when $l = m = 0$ but are consistent with each other since $0^0 = 1$. More generally, let ϕ be a formula with two domains Γ and Δ, and let $n, m \in \mathbb{N}_0$. Then the model count of `Propagate(`ϕ`,` Δ`,` n`)` assuming $|\Gamma| = m$ is the same as the model count of `Propagate(`ϕ`,` Γ`,` m`)` assuming $|\Delta| = n$.

Finally, we note that the `Simplify` procedure plays a crucial role in simplifying a common algebraic pattern $\sum_{m=0}^{n} [a \leq m \leq b] f(m)$. Here: (i) n is a variable, (ii) $a, b \in \mathbb{N}_0$ are constants, (iii) f is an expression that may depend on m, and (iv) $[a \leq m \leq b] = \begin{cases} 1 & \text{if } a \leq m \leq b \\ 0 & \text{otherwise} \end{cases}$ is the Iverson bracket. `Simplify` transforms this pattern into $f(a) + f(a+1) + \cdots + f(\min\{n, b\})$. For instance, in the case of Example 2, `Simplify` transforms $g(l, m) = \sum_{k=0}^{m} [0 \leq k \leq 1] \binom{m}{k} g(l-1, m-k)$ into the simpler form above.

3.1 Identifying a Sufficient Set of Base Cases

Algorithm 2 summarises the implementation of `FindBaseCases`. For each recursive call from a function f to itself, we consider two types of arguments: (i) constants and (ii) arguments of the form $x_i - c_i$, where $c_i \in \mathbb{N}_0$ is a constant, and x_i is the i-th argument of the signature of f. In the former case, we consider a base case specifically for that constant. In the latter case, we consider a base case for constants from zero up to (but not including) c_i. Theorem 1 below motivates this approach.

Example 3. *Consider the recursive function g from Example 2.* `FindBaseCases(`$\mathcal{E}$`)` *iterates over two function calls: $g(l-1, m)$ and $g(l-1, m-1)$. The former produces the base case $g(0, m)$, while the latter produces both $g(0, m)$ and $g(l, 0)$.*

TODO: Prove (lots of notes, handwritten and elsewhere)

Theorem 1 (Termination)**.** *Let us assume the following about the input set of equations $\mathcal{E}$:*

1. *Each function f has exactly one equation with f on the left-hand side. We call this equation the* definition *of f.*

Algorithm 3: `Propagate`(ϕ, Δ, n)

Input: formula ϕ, domain Δ, $n \in \mathbb{N}_0$
Output: formula ϕ'

```
1  φ' ← ∅;
2  if n = 0 then
3      foreach clause C ∈ φ do
4          if Δ ∉ Doms(C) then φ' ← φ' ∪ { C };
5          else
6              C' ← { l ∈ C | Δ ∉ Doms(l) };
7              if C' ≠ ∅ then
8                  l ← an arbitrary literal in C';
9                  φ' ← φ' ∪ { C' ∪ { ¬l } };
10 else
11     D ← a set of n new constants in Δ;
12     foreach clause C ∈ φ do
13         (x_i)_{i=1}^m ← the variables in C with domain Δ;
14         if m = 0 then φ' ← φ' ∪ { C };
15         else
16             φ' ← φ' ∪ { C[x_1 ↦ c_1, ..., x_m ↦ c_m] |
                   (c_i)_{i=1}^m ∈ D^m };
```

2. *There exists a topological ordering of all functions $(f_i)_i$ such that the definition of f_i does not contain function calls to f_j with $j > i$.*[5]
3. *For every equation $(f(\mathbf{x}) = \texttt{expr}) \in \mathcal{E}$, every recursive function call $f(\mathbf{y}) \in \texttt{expr}$ satisfies the following:*
   - *each y_i is either $x_i - c_i$ or c_i for some constant $c_i \in \mathbb{N}_0$;*
   - *there exists i such that $y_i = x_i - c_i$ for some $c_i > 0$.*

Then the set of base cases $\mathcal{B}$ returned by `FindBaseCases`$(\mathcal{E})$ *is* sufficient *for $\mathcal{E}$ in the following sense. The evaluation of any function call with non-negative arguments terminates as long as the evaluation of base cases is prioritised over the corresponding recursive definitions.*[6]

3.2 Propagating Domain Size Assumptions

`Propagate` (Algorithm 3) modifies formula ϕ with the assumption that domain Δ has size $n \in \mathbb{N}_0$. In the case of $n = 0$, many clauses become vacuously satisfied and can be removed. In the case of $n > 0$[7], we perform partial grounding, using constants to replace all variables quantified over Δ. Algorithm 3 considers these two cases separately. For a literal or a clause C, we write $\mathrm{Doms}(C)$ to denote the set of corresponding domains.

In the case of $n = 0$, consider three types of clauses: (i) those that do not mention Δ, (ii) those in which every literal contains variables quantified over Δ, and (iii) those that have some literals with variables quantified over Δ and some without. Step (i) clauses are transferred to the new formula ϕ' unchanged. For Step (ii) clauses, $C' = \emptyset$, so these clauses are filtered out. One might think that the same should be done with Step (iii) clauses, however, lines 8 and 9 perform a new kind of smoothing, the explanation of which we defer to Section 4.1.

In the case of $n > 0$, we introduce n new constants. Consider an arbitrary clause $C \in \phi$ and let $m \in \mathbb{N}_0$ be the number of variables in C quantified over Δ. If $m = 0$, then, similarly to the previous case, we add C directly to ϕ'. Otherwise, we add a clause to ϕ' with every possible way of replacing the m variables in C with some combination of the n new constants.

Example 4. *Consider the clause $C \equiv \forall x \in \Gamma.\, \forall y, z \in \Delta.\, \neg P(x, y) \vee \neg P(x, z) \vee y = z$. Then $\mathrm{Doms}(C) = \mathrm{Doms}(\neg P(x, y)) = \mathrm{Doms}(\neg P(x, z)) = \{\Gamma, \Delta\}$, and $\mathrm{Doms}(y = z) = \{\Delta\}$. A call to* `Propagate`$(\{C\}, \Delta, 3)$ *would produce the following formula with nine clauses:*

$$\forall x \in \Gamma.\, \neg P(x, c_1) \vee \neg P(x, c_1) \vee c_1 = c_1$$
$$\forall x \in \Gamma.\, \neg P(x, c_1) \vee \neg P(x, c_2) \vee c_1 = c_2$$
$$\vdots$$
$$\forall x \in \Gamma.\, \neg P(x, c_3) \vee \neg P(x, c_3) \vee c_3 = c_3,$$

where c_1, c_2, and c_3 are the new constants.

4 Smoothing

Smoothness originates in propositional knowledge compilation where it is defined as the property that, for every disjunction node, all disjuncts (as subtrees of the circuit) contain the same atoms (Darwiche 2001). Van den Broeck et al. (2011) generalise smoothness to first-order logic, adding set-disjunction and inclusion-exclusion nodes alongside disjunction.

The motivation for smoothing is as follows. Whenever compilation rules such as unit propagation and inclusion-exclusion simplify the formula, some ground atoms might be eliminated from consideration (see Example 5 below). To account for them during counting, smoothing nodes (i.e., tautological clauses such as $P(c) \vee \neg P(c)$) are added to the FCG at an appropriate location.

The rest of this section presents an extension to the smoothing algorithm of Van den Broeck et al. (2011). Section 4.1 describes the role smoothing plays in the base-case-finding algorithm from Section 3. Then, Section 4.2 shows how to adapt smoothing to the compilation rules introduced by Dilkas and Belle (2023).

4.1 Smoothing for Base Cases

In this section, we motivate and describe lines 8 and 9 of Algorithm 3. Suppose that `Propagate` is called with arguments $(\phi, \Delta, 0)$, i.e., we are simplifying formula ϕ by assuming that domain Δ is empty. Informally, if there is a predicate P in ϕ that has nothing to do with domain Δ, the role of smoothing is to preserve all occurrences of P even if

[5] This condition excludes the possibility of mutual recursion and similar cyclic scenarios and is akin to stratified logic programs (Lloyd 1987).

[6] Recall that, as previously discussed, the order in which base cases are considered is immaterial.

[7] None of the formulas considered in this work had $n > 1$.

all clauses with P become vacuously satisfied. We note that the approach presented in this section is far from unique and explain it via an example below.

Example 5. *Let ϕ be*

$$\forall x \in \Delta.\ \forall y, z \in \Gamma.\ Q(x) \vee P(y, z) \tag{3}$$

$$\forall y, z \in \Gamma'.\ P(y, z), \tag{4}$$

where $\Gamma' \subseteq \Gamma$ is a domain introduced by a compilation rule. Note that, as a relation, $P \subseteq \Gamma \times \Gamma$.

Let us reason by hand about the model count of ϕ when $\Delta = \emptyset$. Predicate Q can only take one value, i.e., $Q = \emptyset$. The value of P is fixed over $\Gamma' \times \Gamma'$ by Clause (4) *but is allowed to vary freely over $(\Gamma \times \Gamma) \setminus (\Gamma' \times \Gamma')$ since Clause* (3) *is vacuously satisfied by all structures. Hence, the right model count should be $2^{|\Gamma|^2 - |\Gamma'|^2}$.*

However, without line 9, `Propagate` *would simplify ϕ to $\forall y, z \in \Gamma'.\ P(y, z)$. Here, P is defined as a subset of $\Gamma' \times \Gamma'$. Clearly, this simplified formula has only one model: $\{\, P(y, z) \mid y, z \in \Gamma' \,\}$.*

With line 9 included, ϕ is transformed to

$$\forall y, z \in \Gamma.\ P(y, z) \vee \neg P(y, z)$$

$$\forall y, z \in \Gamma'.\ P(y, z),$$

which retains the correct model count.

Note that the choice of l on line 8 of Algorithm 3 is inconsequential because any choice achieves the same goal: constructing a tautological clause that retains the literals in C'.

4.2 Smoothing the FCG

Smoothing is a two-step process. First, positive unit clauses (denoting sets of ground atoms that are accounted for in the FCG) are propagated 'upwards', i.e., in the opposite direction of FCG arcs. Then, at nodes of certain types, missing atoms are detected and additional nodes are added to account for them. In this section, we: (i) describe the relevant node types from previous work, (ii) show how smoothing ought to work for these node types, and (iii) illustrate how the new smoothing techniques work on two example FCGs.

Before describing the proposed changes to smoothing, we briefly review the compilation rules (and their corresponding node types) introduced by Dilkas and Belle (2023) while referring to the aforementioned paper for precise definitions. *Domain recursion* selects a domain Δ, introduces a new constant $c \in \Delta$, and, for each variable x quantified over Δ, considers two possibilities: $x = c$ or $x \neq c$, modifying the formula accordingly. We denote the resulting node as $\mathrm{DR}(c \in \Delta)$. *Constraint removal* applies to formulas such that, for some domain Δ, each variable x quantified over Δ is followed by the inequality constraint $x \neq c$. In other words, the relevant clauses can be rewritten to begin with $\forall x \in \Delta.\ x \neq c \Rightarrow \ldots$ For such formulas, constraint removal replaces Δ with a new domain Δ' and removes the $x \neq c$ constraints. We denote the resulting node as $\mathrm{CR}(\Delta' \leftarrow \Delta \setminus \{\, c \,\})$. Finally, *caching* detects when the input formula ϕ is equal to a previously-encountered formula ψ except for having different domains. We denote the resulting node as $\mathrm{Ref}(\sigma)$, where σ is the substitution mapping the domains of ψ to their corresponding domains in ϕ.

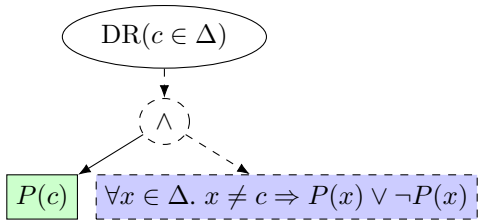

Figure 2: An artificial example of an FCG where a smoothing node must be added below a domain recursion node. The dashed nodes and arcs are added during Stage 2 of smoothing.

Stage 1 for Domain Recursion When visiting a $\mathrm{DR}(c \in \Delta)$ node, for each unit clause received from the child node, we replace each occurrence of $\phi(c)$ or $\forall x \in \Delta.\ x \neq c \Rightarrow \phi(x)$ with $\forall x \in \Delta.\ \phi(x)$. In other words, if the subgraph below the domain recursion node covers some of the ground atoms that are affected by domain recursion, then it should cover all of them. If the relevant subgraph indeed covers those ground atoms, Stage 2 will do nothing. Otherwise, smoothing nodes will be added below the domain recursion node to cover the difference between the sets of ground atoms assigned to the domain recursion node and its child node.

Stage 1 for Constraint Removal and Caching When visiting a $\mathrm{CR}(\Delta' \leftarrow \Delta \setminus \{\, c \,\})$ node, we reverse constraint removal. In other words, we replace each $\forall x \in \Delta'.\ \phi(x)$ with $\forall x \in \Delta.\ x \neq c \Rightarrow \phi(x)$. When visiting a $\mathrm{Ref}(\sigma)$ node, we apply substitution σ to the unit clauses coming from the child node.

Stage 2 for Domain Recursion We need not add smoothing nodes immediately below constraint removal or caching nodes. However, for domain recursion we do the following.

1. Suppose the set of unit clauses assigned to the child node during Stage 1 contains both $\phi(c)$ and $\forall x \in \Delta.\ x \neq c \Rightarrow \phi(x)$. In other words, the only difference between the two clauses is that one has the constant c whereas the other one has a variable $x \neq c$. In such a case, we merge the two clauses into $\forall x \in \Delta.\ \phi(x)$.
2. If necessary, we add smoothing nodes below the domain recursion node to cover the difference between the unit clauses assigned to the domain recursion node during Stage 1 and the unit clauses of the child node post-processed by the step above.

Example 6 (An FCG that needs smoothing)**.** *Figure 2 shows an FCG with a domain recursion node $\mathrm{DR}(c \in \Delta)$ immediately followed by a single $P(c)$ node. In this case, Stage 1 of smoothing assigns $\{\, \forall x \in \Delta.\ P(x) \,\}$ to the former node and $\{\, P(c) \,\}$ to the latter. Since these two sets of unit clauses cover different ground atoms, Stage 2 adds a smoothing node to cover $P(x)$ for all $x \in \Delta \setminus \{\, c \,\}$.*

Example 7 (A smooth FCG)**.** *Stage 1 of smoothing is more involved in the case of the FCG in Figure 3. The unit clause $P(c)$ propagates to the conjunction node and is then generalised to $\forall x \in \Delta.\ P(x)$ by the domain recursion node.*

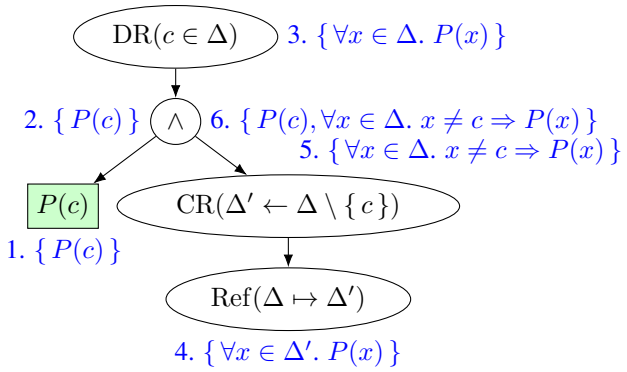

Figure 3: A smooth FCG based on Example 2. The arc from the caching node to the domain recursion node is omitted for compactness. The labels next to nodes show the sets of unit clauses assigned to each node during Stage 1 and the order of these assignments, omitting empty assignments that replace a set S with S itself.

It then propagates to the caching node, changing form to $(\forall x \in \Delta.\ P(x))[\Delta \mapsto \Delta'] \equiv \forall x \in \Delta'.\ P(x)$. *The constraint removal node re-introduces the constraints, transforming the clause to* $\forall x \in \Delta.\ x \neq c \Rightarrow P(x)$, *which is then propagated to the conjunction node, joining* $P(c)$.

In Stage 2, $P(c)$ *and* $\forall x \in \Delta.\ x \neq c \Rightarrow P(x)$ *are merged into* $\forall x \in \Delta.\ P(x)$. *Since the resulting clause matches the clause assigned to the domain recursion node, the FCG is already smooth.*

The algebraic interpretation of Figure 3 is an equation e that defines a recursive function. The right-hand side of e already covers $P(c)$, and the smoothing algorithm correctly recognises that the recursive call also covers $P(x)$ for all $x \in \Delta \setminus \{\, c \,\}$.

5 Generating C++ Code

In this section, we describe the last step of CRANE2 outlined in Figure 1: translating the set of equations $\mathcal{E}$ produced by `CompileWithBaseCases` into C++ code. The produced C++ program can then be compiled and executed with different command-line arguments to compute the model count of the formula for different (combinations of) domain sizes.

Each equation in $\mathcal{E}$ is compiled into a C++ function, together with its own cache for memoisation. Let $e = (f(\mathbf{x}) = \texttt{expr}) \in \mathcal{E}$ be an arbitrary equation, and let $\mathbf{c} \in \mathbb{N}_0^n$ represent the arguments of the corresponding C++ function. The implementation of e consists of three parts. First, we check whether $\mathbf{c}$ is already in the cache of e (in which case the cached value is returned). Second, for each base case $f(\mathbf{y})$ of $f(\mathbf{x})$ (as in Definition 3), we check whether $\mathbf{c}$ *matches* $\mathbf{y}$, i.e., $c_i = y_i$ whenever $y_i \in \mathbb{N}_0$. In this case, $\mathbf{c}$ is redirected to the C++ function that corresponds to the definition of base case $f(\mathbf{y})$. Finally, if the above cases fail, we evaluate $\mathbf{c}$ according to `expr`, store the result in the cache, and return it.

6 Experimental Evaluation

TODO: Introduce greedy search and breadth-first search (BFS), together with CRANE2-GREEDY and CRANE2-BFS. The knowledge compilation part of both CRANE and CRANE2 can be executed using either greedy (similar to FORCLIFT) or breadth-first search. We use both in our experiments, denoting them as CRANE2-GREEDY and CRANE2-BFS, respectively.

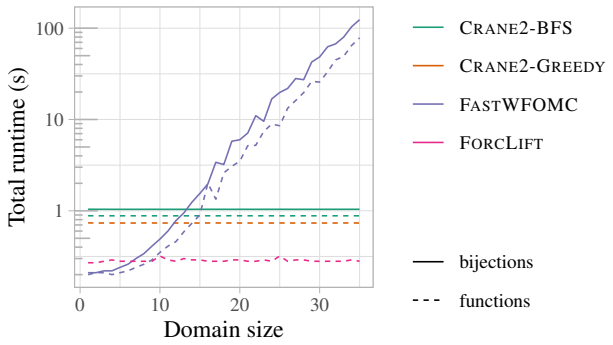

Figure 4: The runtime of WFOMC algorithms. Note that the y axis is on a logarithmic scale.

We compare CRANE2 (in both BFS and greedy modes) with FASTWFOMC and FORCLIFT on two problems previously considered by Dilkas and Belle (2023): the function-counting problem from Example 1 and the bijection-counting problem described below. Note that comparing CRANE2 and FASTWFOMC on a larger set of benchmarks is challenging because there is no automated way to translate a formula in FO or C^2 into $\mathsf{UFO}^2 + \mathsf{EQ}$ (or even check if such an encoding is possible). The experiments were run on an AMD Ryzen 7 5800H processor with 16 GiB of memory and Arch Linux 6.8.2-arch2-1 operating system. FASTWFOMC was executed using Python 3.8.19 with Python-FLINT 0.5.0.

The FO formula for bijections is described in Example 2. The equivalent formula in C^2 is

$$\forall x \in \Delta.\ \exists^{=1} y \in \Delta.\ P(x, y)$$
$$\forall y \in \Delta.\ \exists^{=1} x \in \Delta.\ P(x, y).$$

Similarly, in $\mathsf{UFO}^2 + \mathsf{EQ}$ the same formula can be written as

$$\forall x, y \in \Delta.\ R(x) \vee \neg P(x, y)$$
$$\forall x, y \in \Delta.\ S(x) \vee \neg P(y, x)$$
$$|P| = |\Delta|,$$

where $w^-(R) = w^-(S) = -1$.

In the first experiment, we run each of the four algorithms once on each combination of benchmark and domain size, the latter ranging from one up to and including 35. Figure 4 shows total runtime values as a measure of the overall performance of each algorithm. For the three knowledge compilation algorithms, we also separately track compilation and inference times and comment on them where relevant. Note that CRANE2-BFS is able to handle more instances than either FORCLIFT or CRANE2-GREEDY, i.e., the bijection-counting problem in our experiments as well as similar formulas examined previously (Dilkas and Belle 2023).

Algorithm	Bijections	Functions
CRANE2-BFS	10^4	3×10^5
CRANE2-GREEDY	—	3×10^5
FASTWFOMC	28	32
FORCLIFT	—	143

Table 2: The maximum domain sizes that can be handled by each algorithm in at most $45\,\mathrm{s}$. '—' indicates that the algorithm is unable to find a solution. The domain sizes for CRANE2 are accurate up to the first digit while other domain sizes are exact.

As shown in Figure 4, the runtimes of all compilation-based algorithms remain practically constant in contrast to the rapidly increasing runtimes of FASTWFOMC. Although the search/compilation part is slower in CRANE2 than in FORCLIFT, the difference is negligible. The runtimes of the knowledge compilation algorithms appear constant because—for these counting problems and domain sizes—compilation time dominates inference time (recall that compilation time is independent of domain sizes). Indeed, the maximum inference time of both CRANE2-BFS and CRANE2-GREEDY across these experiments is only $4\,\mathrm{ms}$. The runtimes of CRANE2 have lower variation than those of FORCLIFT because with FORCLIFT we compile the formula anew for each domain size whereas with CRANE2 we compile it once and reuse the resulting C++ program for all domain sizes.

For the second experiment, we examine the maximum domain size that the algorithms can handle in at most $45\,\mathrm{s}$ of total runtime. As shown in Table 2, CRANE2-BFS is able to scale to domain sizes 357 and 9375 times larger than FASTWFOMC for bijection-counting and function-counting problems, respectively. Note that, while FORCLIFT may have runtime performance comparable to CRANE2, its finite-precision arithmetic cannot represent model counts for domain sizes higher than 143 and returns ∞ instead.

7 Conclusion

TODO

References

Azzolini, D., and Riguzzi, F. 2023. Lifted inference for statistical statements in probabilistic answer set programming. *Int. J. Approx. Reason.* 163:109040.

Barvínek, J.; van Bremen, T.; Wang, Y.; Zelezný, F.; and Kuželka, O. 2021. Automatic conjecturing of P-recursions using lifted inference. In *ILP*, volume 13191 of *Lecture Notes in Computer Science*, 17–25. Springer.

Beame, P.; Van den Broeck, G.; Gribkoff, E.; and Suciu, D. 2015. Symmetric weighted first-order model counting. In *PODS*, 313–328. ACM.

Darwiche, A. 2001. On the tractable counting of theory models and its application to truth maintenance and belief revision. *Journal of Applied Non-Classical Logics* 11(1-2):11–34.

Dilkas, P., and Belle, V. 2023. Synthesising recursive functions for first-order model counting: Challenges, progress, and conjectures. In *KR*, 198–207.

Gogate, V., and Domingos, P. M. 2016. Probabilistic theorem proving. *Commun. ACM* 59(7):107–115.

Gribkoff, E.; Suciu, D.; and Van den Broeck, G. 2014. Lifted probabilistic inference: A guide for the database researcher. *IEEE Data Eng. Bull.* 37(3):6–17.

Hinman, P. G. 2018. *Fundamentals of mathematical logic*. CRC Press.

Jaeger, M., and Van den Broeck, G. 2012. Liftability of probabilistic inference: Upper and lower bounds. In *StarAI@UAI*.

Kazemi, S. M., and Poole, D. 2016. Knowledge compilation for lifted probabilistic inference: Compiling to a low-level language. In *KR*, 561–564. AAAI Press.

Kazemi, S. M.; Kimmig, A.; Van den Broeck, G.; and Poole, D. 2016. New liftable classes for first-order probabilistic inference. In *NIPS*, 3117–3125.

Kersting, K. 2012. Lifted probabilistic inference. In *ECAI*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, 33–38. IOS Press.

Kuželka, O. 2021. Weighted first-order model counting in the two-variable fragment with counting quantifiers. *J. Artif. Intell. Res.* 70:1281–1307.

Lloyd, J. W. 1987. *Foundations of Logic Programming, 2nd Edition*. Springer.

Malhotra, S., and Serafini, L. 2022. Weighted model counting in FO2 with cardinality constraints and counting quantifiers: A closed form formula. In *AAAI*, 5817–5824. AAAI Press.

Niu, F.; Ré, C.; Doan, A.; and Shavlik, J. W. 2011. Tuffy: Scaling up statistical inference in Markov logic networks using an RDBMS. *Proc. VLDB Endow.* 4(6):373–384.

Riguzzi, F.; Bellodi, E.; Zese, R.; Cota, G.; and Lamma, E. 2017. A survey of lifted inference approaches for probabilistic logic programming under the distribution semantics. *Int. J. Approx. Reason.* 80:313–333.

Svatos, M.; Jung, P.; Tóth, J.; Wang, Y.; and Kuželka, O. 2023. On discovering interesting combinatorial integer sequences. In *IJCAI*, 3338–3346. ijcai.org.

Tóth, J., and Kuželka, O. 2023. Lifted inference with linear order axiom. In *AAAI*, 12295–12304. AAAI Press.

Totis, P.; Davis, J.; De Raedt, L.; and Kimmig, A. 2023. Lifted reasoning for combinatorial counting. *J. Artif. Intell. Res.* 76:1–58.

van Bremen, T., and Kuželka, O. 2020. Approximate weighted first-order model counting: Exploiting fast approximate model counters and symmetry. In *IJCAI*, 4252–4258. ijcai.org.

van Bremen, T., and Kuželka, O. 2021. Faster lifting for two-variable logic using cell graphs. In *UAI*, volume 161 of *Proceedings of Machine Learning Research*, 1393–1402. AUAI Press.

van Bremen, T., and Kuželka, O. 2023. Lifted inference with tree axioms. *Artif. Intell.* 324:103997.

Van den Broeck, G.; Taghipour, N.; Meert, W.; Davis, J.; and De Raedt, L. 2011. Lifted probabilistic inference by

first-order knowledge compilation. In *IJCAI*, 2178–2185. IJCAI/AAAI.

Van den Broeck, G.; Meert, W.; and Darwiche, A. 2014. Skolemization for weighted first-order model counting. In *KR*. AAAI Press.

Van den Broeck, G. 2011. On the completeness of first-order knowledge compilation for lifted probabilistic inference. In *NIPS*, 1386–1394.

Venugopal, D.; Sarkhel, S.; and Gogate, V. 2015. Just count the satisfied groundings: Scalable local-search and sampling based inference in mlns. In *AAAI*, 3606–3612. AAAI Press.

Wang, Y.; van Bremen, T.; Wang, Y.; and Kuželka, O. 2022. Domain-lifted sampling for universal two-variable logic and extensions. In *AAAI*, 10070–10079. AAAI Press.

Wang, Y.; Pu, J.; Wang, Y.; and Kuželka, O. 2023. On exact sampling in the two-variable fragment of first-order logic. In *LICS*, 1–13.