Towards Practical First-Order Model Counting

Anonymous submission

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. 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 500 000 times larger than the current state of the art, as demonstrated through experimental results.

1 Introduction

First-order model counting is the task of counting the number of models of a sentence in first-order logic over some given domain(s). The (symmetric) weighted variant of this problem, known as WFOMC, seeks to calculate the sum of model weights. In WFOMC, the weight of a model is determined by predicate weights (Van den Broeck et al. 2011). WFOMC is a key approach to *lifted (probabilistic) inference*, which aims to compute probabilities more efficiently by leveraging symmetries inherent in the problem description (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 has been used for inference on probabilistic databases (Gribkoff, Suciu, and Van den Broeck 2014) and probabilistic logic programs (Riguzzi et al. 2017). By considering domains of increasing sizes, the model count of a formula can be seen as an integer sequence. WFOMC algorithms have been utilised for discovering new sequences (Svatos et al. 2023) as well as conjecturing (Barvínek et al. 2021) and constructing (Dilkas and Belle 2023) recurrence relations and other recursive structures that describe these sequences. Additionally, WFOMC algorithms have been extended to perform *sampling* (Wang et al. 2022, 2023).

The complexity of WFOMC is typically characterised in terms of *data complexity*. This involves fixing the formula and determining whether an algorithm exists that can compute the WFOMC in time polynomial with respect to the domain size(s). If such an algorithm exists, the formula is called *liftable* (Jaeger and Van den Broeck 2012). Beame et al. (2015) demonstrated the existence of 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). The liftable fragment of formulas with two variables has been expanded 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 are various WFOMC algorithms with different underlying principles. Perhaps the most prominent class of WFOMC algorithms is based on *first-order knowledge compilation* (FOKC). In this approach, the formula is compiled into a representation (such as a circuit or graph) by iteratively applying *compilation rules*. This representation 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, while the latter compiles them first to graphs and then to (algebraic) equations. Another WFOMC algorithm, FASTWFOMC (van Bremen and Kuželka 2021), is based on cell enumeration. Other algorithms utilise 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 capable of handling formulas beyond the reach of FORCLIFT, was 'incomplete' as it could only construct function definitions, which would then need to be evaluated to compute the WFOMC. In other words, it could not be used as a black box that takes Markov logic networks (or weighted formulas in some other format) and outputs numbers (i.e., model counts or probabilities). Additionally, recursive functions were presented without base cases. In this work, we introduce CRANE2, an extension of CRANE that addresses 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 corresponding to these base cases, and (iii) recursing on these new formulas. Then, Section 4 explains post-processing techniques for the formulas from Step (ii) above required to preserve the correct model count. Next, Section 5 elucidates how function definitions encoding a solution to a WFOMC problem are compiled into C++ programs. Note that a solution to WFOMC that uses compilation to C++ has been considered before (Kazemi and Poole 2016), however, the extent of formulas that could be handled was limited. Finally, Section 6 presents experiments results comparing 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 review the basic concepts of first-order logic as they are used in FOKC algorithms. There are two key differences between the logic used by 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. Note that Skolemization here is different from the standard Skolemization procedure that introduces function symbols (Hodges 1997). Second, the input formula is rewritten as a conjunction of clauses, each of which is in *prenex normal form* (Hinman 2018).

A *term* is either a variable or a constant. 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. The *arity* of a predicate is the number of arguments it takes, i.e., m in the case of predicate P above. We write P/m to denote a predicate together with its arity. A *literal* is either an atom (i.e., a *positive* literal) or its negation (i.e., a *negative* literal). An atom is *ground* if it contains no variables, i.e., only constants. 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. Finally, a *formula* is a conjunction of clauses. Throughout the paper, we use set-theoretic notation, interpreting a formula as a set of clauses and a clause as a set of literals.

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[*] and its extensions CRANE[†] and CRANE2; (ii) C^2 is often used in the literature on FASTWFOMC and related methods (Kuželka 2021; Malhotra and Serafini 2022); (iii) $\mathsf{UFO}^2 + \mathsf{CC}$ is the input format supported by a private version of FASTWFOMC obtained directly from the authors. Note that the publicly available version[‡] 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 and redefined to be more specific in the case of the other two logics. All three logics are function-free, and domains are always assumed to be finite. As usual, we presuppose the *unique name assumption*, which states that two constants are equal if and only if they are the same constant (Russell and Norvig 2020).

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. 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, $\forall x \in \Delta.\ \forall y \in \Gamma.\ P(x, y) \lor x = y$. FO is also the only logic to support constants, formulas with more than two variables, and the equality predicate. While we do not explicitly refer to sorts in subsequent sections of this paper, the many-sorted nature of FO is paramount to the algorithms presented therein.

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).

Compared to FO, C^2 and $\mathsf{UFO}^2 + \mathsf{CC}$ lack support for: (i) constants, (ii) the equality predicate, (iii) multiple domains, and (iv) formulas with more than two variables. The advantage that C^2 brings over FO is the inclusion of *counting quantifiers*. That is, alongside $\forall$ and $\exists$, C^2 supports $\exists^{=k}$, $\exists^{\le k}$, and $\exists^{\ge k}$ for any positive integer k. For example, $\exists^{=1} x.\ \phi(x)$ means that there exists *exactly one* x such that $\phi(x)$, and $\exists^{\le 2} x.\ \phi(x)$ means that there exist *at most two* such x. $\mathsf{UFO}^2 + \mathsf{CC}$, on the other hand, does not support any existential quantifiers but instead incorporates *(equality) cardinality constraints*. For example, $|P| = 3$ constrains all models to have *precisely three positive literals with the predicate* P.

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

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

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

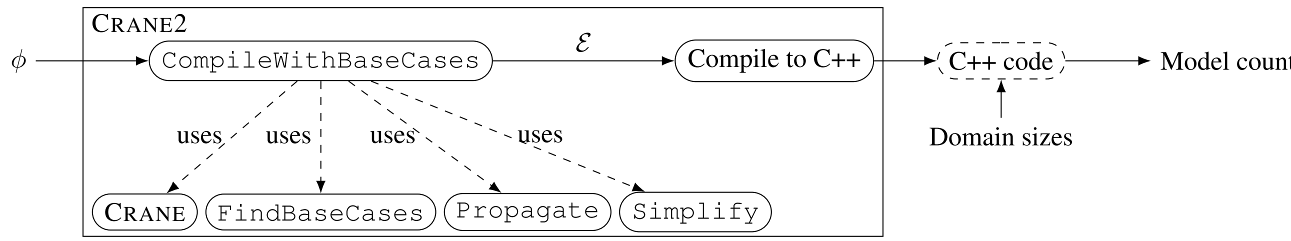

Figure 1: The outline of using CRANE2 to compute the model count of a formula ϕ. First, the formula is compiled into a set of equations, which are then used to create a C++ program. This program can be executed with different command line arguments to calculate the model count of ϕ for different domain sizes. To accomplish this, the `CompileWithBaseCases` function employs several components: (i) the FOKC algorithm of CRANE, (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	$\forall$, $\exists$, $\exists^{=k}$, $\exists^{\leq k}$, $\exists^{\geq k}$	—
$\mathsf{UFO}^2 + \mathsf{CC}$	one	✗	two	$\forall$	$\lvert P\rvert = m$

Table 1: A comparison of the three logics used in WFOMC based on the following aspects: (i) the number of sorts, (ii) support for constants, (iii) the 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 $\mathbf{t}$. Here: (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.

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 the domains of ϕ to their interpretations as sets, satisfying the following conditions: (i) the sets are pairwise disjoint, and (ii) the constants in ϕ are included in the corresponding domains. (In practice, we typically only specify the size of each domain.) A *structure* of ϕ (with respect to σ) is a set M of ground literals defined by adding to M 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 a *model* if it satisfies ϕ.

Remark. The distinctness of domains is important in two ways. First, in terms of expressiveness, clauses such as $\forall x \in \Delta.\ P(x, x)$ are valid if predicate P is defined over two copies of the same domain and invalid otherwise. Second, having more distinct domains makes the problem more decomposable for the FOKC algorithm. With more distinct domains, the algorithm can make assumptions or deductions about, e.g., the first domain of predicate P without worrying how (or if) they apply to the second domain.

Definition 2 (WFOMC Instance)**.** A *WFOMC instance* comprises of: (i) a formula ϕ in FO, (ii) two (rational) *weights* $w^+(P)$ and $w^-(P)$ assigned to each predicate P in ϕ, and (iii) σ as described in Definition 1. Unless specified otherwise, we assume all weights to be equal to 1.

Definition 3 (WFOMC (Van den Broeck et al. 2011))**.** Given a WFOMC instance (ϕ, w^+, w^-, σ) as in Definition 2, the *(symmetric) weighted first-order model count* (WFOMC) of ϕ (with respect to σ, w^+, and w^-) is

$$\sum_{M \models \phi} \prod_{P(\mathbf{t}) \in M} w^+(P) \prod_{\neg P(\mathbf{t}) \in M} w^-(P), \tag{1}$$

where the sum is over all models of ϕ.

Example 1 (Counting functions)**.** To define predicate P as a function from a domain Δ to itself, in C^2 one would write $\forall x \in \Delta.\ \exists^{=1} y \in \Delta.\ P(x, y)$. In $\mathsf{UFO}^2 + \mathsf{CC}$, the same could be written as

$$\begin{gathered}(\forall x, y \in \Delta.\ S(x) \lor \neg P(x, y)) \land \\ (\lvert P\rvert = \lvert\Delta\rvert),\end{gathered} \tag{2}$$

where $w^-(S) = -1$. Although Formula (2) has more models compared to its counterpart in C^2, the negative weight $w^-(S) = -1$ makes some of the terms in Equation (1) cancel out.

Equivalently, in FO we would write

$$\begin{gathered}(\forall x \in \Gamma.\ \exists y \in \Delta.\ P(x, y)) \land \\ (\forall x \in \Gamma.\ \forall y, z \in \Delta.\ P(x, y) \land P(x, z) \Rightarrow y = z).\end{gathered} \tag{3}$$

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 (3) is written with two domains instead of just one. However, we can still determine the correct number of functions 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 FOKC algorithms to find efficient solutions.

Algorithm 1: `CompileWithBaseCases(`ϕ`)`

Input: formula ϕ
Output: set $\mathcal{E}$ of equations
1 $(\mathcal{E}, \mathcal{F}, \mathcal{D}) \leftarrow$ CRANE (ϕ) ;
2 $\mathcal{E} \leftarrow$ `Simplify(`$\mathcal{E}$`)`;
3 **foreach** *base case* $f(\mathbf{x}) \in$ `FindBaseCases(`$\mathcal{E}$`)` **do**
4     $\psi \leftarrow \mathcal{F}(f)$;
5   **foreach** *index* i *such that* $x_i \in \mathbb{N}_0$ **do**
6         $\psi \leftarrow$ `Propagate(`ψ, $\mathcal{D}(f, i)$, x_i`)`;
7     $\mathcal{E} \leftarrow \mathcal{E} \cup$ `CompileWithBaseCases(`ψ`)`;

2.3 Algebra

We write `expr` to represent an arbitrary algebraic expression. It is important to note that some terms have different meanings in 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 $\sum_{i=1}^{n}$ `expr`. A (function) *signature* is $f(x_1, \ldots, x_n)$ (or $f(\mathbf{x})$ for short), where f represents an n-ary function, and each x_i represents a variable. An *equation* is $f(\mathbf{x}) =$ `expr`, with $f(\mathbf{x})$ representing a signature.

Definition 4 (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 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.

3 Completing the Definitions of Functions

Algorithm 1 presents our overall approach for compiling a formula into equations that include the necessary base cases. To begin, we use the FOKC algorithm of the original CRANE to compile the formula into three components: (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 (explained at the end of this section), $\mathcal{E}$ is passed to the `FindBaseCases` procedure (explained in Section 3.1), which identifies the base cases that need to be defined.

For each base case $f(\mathbf{x})$, we retrieve the logical formula $\mathcal{F}(f)$ associated with the function name f and simplify it using the `Propagate` procedure (explained in detail in Section 3.2). We do this by iterating over all indices of $\mathbf{x}$, where x_i is a constant, and using `Propagate` to simplify ψ by assuming that domain $\mathcal{D}(f, i)$ has size x_i. Finally, on line 7, `CompileWithBaseCases` recurses on these simplified formulas and adds the resulting base case equations to $\mathcal{E}$. Example 2 below provides more detail.

Remark. Although `CompileWithBaseCases` starts with a call to CRANE, the proposed algorithm is not just a post-processing step for FOKC because Algorithm 1 is recursive and can issue more calls to CRANE on various derived formulas.

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

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

We specifically examine the first solution returned by CRANE2 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 CRANE. 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 $\forall x \in \Gamma.\ S(x) \lor \neg S(x)$, where S is a predicate introduced by Skolemization with weights $w^{+}(S) = 1$ and $w^{-}(S) = -1$. Hence, we obtain the base case $g(0, m) = 0^m$. In the case of $g(l, 0)$, `Propagate(`ψ, Γ, 0`)` returns an empty formula, resulting in $g(l, 0) = 1$.

It is worth noting that these base cases overlap when $l = m = 0$ but remain consistent since $0^0 = 1$. Generally, let ϕ be a formula with two domains Γ and Δ, and let $n, m \in \mathbb{N}_0$. Then the WFOMC of `Propagate(`ϕ, Δ, n`)` assuming $|\Gamma| = m$ is the same as the WFOMC of `Propagate(`ϕ, Γ, m`)` assuming $|\Delta| = n$.

Finally, we main responsibility of the `Simplify` procedure is in handling the algebraic pattern $\sum_{m=0}^{n} [a \le m \le b] f(m)$. 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) $[a \le m \le b] = \begin{cases} 1 & \text{if } a \le m \le b \\ 0 & \text{otherwise} \end{cases}$. `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 $g(l, m) = \sum_{k=0}^{m} [0 \le k \le 1] \binom{m}{k} g(l-1, m-k)$ into $g(l, m) = g(l-1, m) + m g(l-1, m-1)$.

3.1 Identifying a Sufficient Set of Base Cases

Algorithm 2 summarises the implementation of the `FindBaseCases` function. When a function f calls itself recursively, `FindBaseCases` considers two types of arguments: (i) constants and (ii) arguments of the form $x_i - c_i$, where c_i is a constant and x_i is the i-th argument of the signature of f. If the argument is a constant c_i, a base case with c_i is added. In the second case, a base case is added for each constant from zero up to (but not including) c_i. The following discussion explains the reasoning behind this approach.

Example 3. Consider the recursive function g from Example 2. Then `FindBaseCases` iterates over two function calls: $g(l-1, m)$ and $g(l-1, m-1)$. The former produces

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

Input: set $\mathcal{E}$ of equations
Output: set $\mathcal{B}$ of base cases
$\mathcal{B} \leftarrow \emptyset$;
foreach *function call* $f(\mathbf{y})$ *on the right-hand side of an equation in* $\mathcal{E}$ **do**
  $\mathbf{x} \leftarrow$ the parameters of f in its definition;
  foreach $y_i \in \mathbf{y}$ **do**
    if $y_i \in \mathbb{N}_0$ **then**
      $\mathcal{B} \leftarrow \mathcal{B} \cup \{ f(\mathbf{x})[x_i \mapsto y_i] \}$;
    else if $y_i = x_i - c_i$ *for some* $c_i \in \mathbb{N}_0$ **then**
      for $j \leftarrow 0$ **to** $c_i - 1$ **do**
        $\mathcal{B} \leftarrow \mathcal{B} \cup \{ f(\mathbf{x})[x_i \mapsto j] \}$;

the base case $g(0, m)$, while the latter produces both $g(0, m)$ and $g(l, 0)$.

For the rest of this section, let $\mathcal{E}$ represent the equations returned by `CompileWithBaseCases`. To demonstrate that the base cases identified by `FindBaseCases` are sufficient, we begin with a few observations that stem from the details of previous work (Van den Broeck et al. 2011; Dilkas and Belle 2023) and this work.

Observation 1. *For each function f, there is precisely one equation $e \in \mathcal{E}$ with $f(\mathbf{x})$ on the left-hand side where all x_i's are variables (i.e., e is not a base case). We refer to e as the* definition *of f.*

Observation 2. *There is a* topological ordering *of all functions $(f_i)_i$ in $\mathcal{E}$ such that equations in $\mathcal{E}$ with f_i on the left-hand side do not contain function calls to f_j with $j > i$. This condition prevents mutual recursion and other cyclic scenarios.*

Observation 3. *For every equation $(f(\mathbf{x}) = \texttt{expr}) \in \mathcal{E}$, the evaluation of* `expr` *terminates when provided with the values of all relevant function calls.*

Corollary 1. *If f is a non-recursive function with no function calls on the right-hand side of its definition, then the evaluation of any function call $f(\mathbf{x})$ terminates.*

Observation 4. *For any equation $f(\mathbf{x}) = \texttt{expr}$, if $\mathbf{x}$ contains only constants, then* `expr` *cannot include any function calls to f.*

Additionally, we introduce an assumption about the structure of recursion.

Assumption 1. *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.*
- *There exists i such that $y_i = x_i - c_i$ for some $c_i > 0$.*

Finally, we assume a particular order of evaluation for function calls using the equations in $\mathcal{E}$. Specifically, we assume that base cases are considered before the recursive definition. The exact order in which base cases are considered is immaterial.

Assumption 2. *When multiple equations in $\mathcal{E}$ match a function call $f(\mathbf{x})$, preference is given to an equation with the most constants on its left-hand side.*

With the observations and assumptions mentioned above, we prove the following theorem.

Theorem 1 (Termination)**.** *Let f be an n-ary function in $\mathcal{E}$ and $\mathbf{x} \in \mathbb{N}_0^n$. Then the evaluation of $f(\mathbf{x})$ terminates.*

We prove Theorem 1 using double induction. First, we apply induction on the number of functions in $\mathcal{E}$. Then, we use induction on the arity of the 'last' function in $\mathcal{E}$ according to some topological ordering (as defined in Observation 2). For readability, we divide the proof into several lemmas of increasing generality.

Lemma 1. *Assume that $\mathcal{E}$ consists of just* one unary *function f. Then the evaluation of a function call $f(x)$ terminates for any $x \in \mathbb{N}_0$.*

Proof. If $f(x)$ is captured by a base case, then its evaluation terminates by Corollary 1 and Observation 4. If f is not recursive, the evaluation of $f(x)$ terminates by Corollary 1.

Otherwise, let $f(y)$ be an arbitrary function call on the right-hand side of the definition of $f(x)$. If y is a constant, then there is a base case for $f(y)$. Otherwise, let $y = x - c$ for some $c > 0$. Then there exists $k \in \mathbb{N}_0$ such that $0 \leq x - kc \leq c - 1$. So, after k iterations, the sequence of function calls $f(x), f(x - c), f(x - 2c), \dots$ will be captured by the base case $f(x \mod c)$. □

Lemma 2. *Generalising Lemma 1, let $\mathcal{E}$ be a set of equations for* one *n-ary function f for some $n \geq 1$. Then the evaluation of $f(\mathbf{x})$ terminates for any $\mathbf{x} \in \mathbb{N}_0^n$.*

Proof. If f is non-recursive, the evaluation of $f(\mathbf{x})$ terminates by previous arguments. We proceed by induction on n, with the base case of $n = 1$ handled by Lemma 1. Assume that $n > 1$. Any base case of f can be seen as a function of arity $n - 1$, since one of the parameters is fixed. Thus, the evaluation of any base case terminates by the inductive hypothesis. It remains to show that the evaluation of the recursive equation for f terminates, but that follows from Observation 3. □

Proof of Theorem 1. We proceed by induction on the number of functions n. The base case of $n = 1$ is handled by Lemma 2. Let $(f_i)_{i=1}^n$ be some topological ordering of these $n > 1$ functions. If $f = f_j$ for $j < n$, then the evaluation of $f(\mathbf{x})$ terminates by the inductive hypothesis since f_j cannot call f_n by Observation 2. Using the inductive hypothesis that all function calls to f_j (with $j < n$) terminate, the proof proceeds similarly to the Proof of Lemma 2. □

3.2 Propagating Domain Size Assumptions

Algorithm 3, called `Propagate`, modifies the formula ϕ based on the assumption that $|\Delta| = n$. When $n = 0$, some clauses become vacuously satisfied and can be removed. When $n > 0$, partial grounding is performed by replacing all variables quantified over Δ with constants. (None of the formulas examined in this work had $n > 1$.) Algorithm 3

Algorithm 3: `Propagate`(ϕ, Δ, n)

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

1 $\phi' \leftarrow \emptyset$;
2 **if** $n = 0$ **then**
3 　**foreach** *clause* $C \in \phi$ **do**
4 　　**if** $\Delta \notin \mathrm{Doms}(C)$ **then** $\phi' \leftarrow \phi' \cup \{ C \}$;
5 　　**else**
6 　　　$C' \leftarrow \{ l \in C \mid \Delta \notin \mathrm{Doms}(l) \}$;
7 　　　**if** $C' \neq \emptyset$ **then**
8 　　　　$l \leftarrow$ an arbitrary literal in C';
9 　　　　$\phi' \leftarrow \phi' \cup \{ C' \cup \{ \neg l \} \}$;
10 **else**
11 　$D \leftarrow$ a set of n new constants in Δ;
12 　**foreach** *clause* $C \in \phi$ **do**
13 　　$(x_i)_{i=1}^m \leftarrow$ the variables in C with domain Δ;
14 　　**if** $m = 0$ **then** $\phi' \leftarrow \phi' \cup \{ C \}$;
15 　　**else**
16 　　　$\phi' \leftarrow \phi' \cup \{ C[x_1 \mapsto c_1, \ldots, x_m \mapsto c_m] \mid (c_i)_{i=1}^m \in D^m \}$;

handles these two cases separately. For a literal or a clause C, the set of corresponding domains is denoted as $\mathrm{Doms}(C)$.

In the case of $n = 0$, there are three types of clauses to consider: (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. Clauses of Type (i) are transferred to the new formula ϕ' without any changes. For clauses of Type (ii), C' is empty, so these clauses are filtered out. As for clauses of Type (iii), a new kind of smoothing is performed, which will be explained in Section 4.

In the case of $n > 0$, n new constants are introduced. Let C be an arbitrary clause in ϕ, and let $m \in \mathbb{N}_0$ be the number of variables in C quantified over Δ. If $m = 0$, C is added directly to ϕ'. Otherwise, a clause is added to ϕ' for every possible combination of replacing the m variables in C with the n new constants.

Example 4. Let $C \equiv \forall x \in \Gamma. \forall y, z \in \Delta. \neg P(x, y) \lor \neg P(x, z) \lor y = z$. 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`($\{ C \}$, Δ, 3) would result in the following formula with nine clauses:

$$
\begin{gathered}
(\forall x \in \Gamma. \neg P(x, c_1) \lor \neg P(x, c_1) \lor c_1 = c_1) \land \\
(\forall x \in \Gamma. \neg P(x, c_1) \lor \neg P(x, c_2) \lor c_1 = c_2) \land \\
\vdots \\
(\forall x \in \Gamma. \neg P(x, c_3) \lor \neg P(x, c_3) \lor c_3 = c_3).
\end{gathered}
$$

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

4 Smoothing the Base Cases

Smoothing is the process of modifying a circuit to reintroduce atoms that might have been eliminated, thus ensuring that the circuit preserves the correct (weighted) model count (Darwiche 2001; Van den Broeck et al. 2011). In this section, we motivate and describe a similar process performed on lines 8 and 9 of Algorithm 3. Line 7 checks whether smoothing is necessary, and lines 8 and 9 execute it. If the condition on line 7 is not satisfied, the clause is not smoothed but rather completely omitted.

Suppose that `Propagate` is called with arguments $(\phi, \Delta, 0)$, which means we are simplifying the formula ϕ by assuming that the domain Δ is empty. Informally, if there is a predicate P in ϕ unrelated to Δ, smoothing preserves all occurrences of P even if all clauses with P become vacuously satisfied.

Example 5. Let ϕ be:

$$
(\forall x \in \Delta. \forall y, z \in \Gamma. Q(x) \lor P(y, z)) \land \tag{4}
$$
$$
(\forall y, z \in \Gamma'. P(y, z)), \tag{5}
$$

where $\Gamma' \subseteq \Gamma$ is a domain introduced by a compilation rule. It should be noted that P, as a relation, is a subset of $\Gamma \times \Gamma$.

Now, let us reason manually about the model count of ϕ when $\Delta = \emptyset$. Predicate Q can only take one value, $Q = \emptyset$. The value of P is fixed over $\Gamma' \times \Gamma'$ by Clause (5), but it is allowed to vary freely over $(\Gamma \times \Gamma) \setminus (\Gamma' \times \Gamma')$ since Clause (4) is vacuously satisfied by all structures. Therefore, the correct WFOMC should be $2^{|\Gamma|^2 - |\Gamma'|^2}$.

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

By including line 9, `Propagate` transforms ϕ to:

$$
\begin{gathered}
(\forall y, z \in \Gamma. P(y, z) \lor \neg P(y, z)) \land \\
(\forall y, z \in \Gamma'. P(y, z)),
\end{gathered}
$$

which retains the correct model count.

It is worth mentioning 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'.

5 Generating C++ Code

In this section, we will describe the final step of CRANE2 as outlined in Figure 1. This step involves translating the set of equations $\mathcal{E}$ into C++ code. The resulting C++ program can then be compiled and executed with different command-line arguments to compute the model count of the formula for various domain sizes.

Each equation in $\mathcal{E}$ is compiled into a C++ function, along with a separate cache for memoisation. Let us consider an arbitrary equation $e = (f(\mathbf{x}) = \texttt{expr}) \in \mathcal{E}$, 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 if $\mathbf{c}$ is already present in the cache of e. If it is, we simply return the cached value. Second, for each base case

$f(\mathbf{y})$ of $f(\mathbf{x})$ (as defined in Definition 4), we check if $\mathbf{c}$ *matches* $\mathbf{y}$, i.e., $c_i = y_i$ whenever $y_i \in \mathbb{N}_0$. If this condition is satisfied, $\mathbf{c}$ is redirected to the C++ function that corresponds to the definition of the base case $f(\mathbf{y})$. Finally, if none of the above cases apply, we evaluate $\mathbf{c}$ based on the expression `expr`, store the result in the cache, and return it.

6 Experimental Evaluation

In this section, we present experimental results examining the scalability of CRANE2 compared to two other notable FOMC algorithms: FASTWFOMC and FORCLIFT. Note that it would not make sense to include the original CRANE algorithm in the experiments or to evaluate CRANE2 without the C++ code generation described in Section 5. Indeed, these algorithms can only produce mathematical definitions of functions without evaluating them.

In our experiments, we include two versions of CRANE2: CRANE2-GREEDY and CRANE2-BFS. Like its predecessor, CRANE2 has two modes that decide how compilation rules are applied to formulas: one that uses greedy search similarly to FORCLIFT and another that combines greedy and breadth-first search. While greedy search is more efficient (in terms of being able to handle larger formulas), it is a heuristic that can miss some solutions.

We compare the algorithms on three benchmarks from previous work. The first is the function-counting problem from Example 1, previously examined by Dilkas and Belle (2023). The second is a variant of the well-known 'Friends & Smokers' Markov logic network (Singla and Domingos 2008; Van den Broeck, Choi, and Darwiche 2012). In C^2, FO, and $\mathsf{UFO}^2 + \mathsf{CC}$, it can be formulated as

$$(\forall x, y \in \Delta.\ S(x) \land F(x, y) \Rightarrow S(y)) \land \\ (\forall x \in \Delta.\ S(x) \Rightarrow C(x))$$

or, equivalently, in conjunctive normal form as

$$(\forall x, y \in \Delta.\ S(y) \lor \neg S(x) \lor \neg F(x, y)) \land \\ (\forall x \in \Delta.\ C(x) \lor \neg S(x)).$$

Finally, we include the bijection-counting problem previously used by Dilkas and Belle (2023). Its formulation in FO is described in Example 2. The equivalent formula in C^2 is

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

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

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

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

Since FASTWFOMC does not support many-sorted logic, our experiments are restricted to formulas with a single domain. However, many of the counting problems examined in the experimental evaluation of CRANE (Dilkas and Belle 2023) become equivalent (in the sense that they generate the same integer sequences) when restricted to a single domain. Moreover, it is challenging to compare CRANE2 and FASTWFOMC on a larger set of problems because each problem has to be represented in two (fairly different) logics: FO and $\mathsf{UFO}^2 + \mathsf{CC}$. Nonetheless, the three benchmarks cover a diverse range of possibilities. Unlike the other two, the 'friends' benchmark uses multiple predicates and can be written in FO using only two variables and no cardinality constraints or counting quantifiers. Next, the 'functions' benchmark, while still liftable by all of the algorithms, requires either the use of cardinality constraints, counting quantifiers, or more than two variables. Finally, the 'bijections' benchmark is an example of a formula that can be handled by FASTWFOMC but not FORCLIFT.

A major difference between FORCLIFT and the other two algorithms is that FORCLIFT does not support arbitrary-precision arithmetic whereas the other two algorithms both use the GNU Multiple Precision Arithmetic Library. Instead, when the numerical value of the count becomes too big to be accurately presented using finite precision, FORCLIFT returns ∞. We run each algorithm on each benchmark on domains of size $2^1, 2^2, 2^3, \ldots$ until the algorithm times out after $1\,\mathrm{h}$, runs out of memory, or (in the case of FORCLIFT) returns ∞. We measure compilation and inference time separately, but primarily focus on total runtime (i.e., the sum of both), which is dominated by the latter.

The experiments were run on an Intel Skylake $2.4\,\mathrm{GHz}$ CPU with $188\,\mathrm{GiB}$ of memory and CentOS 7 operating system. C++ programs were compiled using the Intel C++ Compiler 2020u4. FASTWFOMC was run on Julia 1.10.4, while all other algorithms were run on the Java Virtual Machine 1.8.0_201.

Figure 2 summarises the experimental results. Only FASTWFOMC and CRANE2-BFS could lift the bijection-counting problem. On this benchmark, the largest domain sizes that these algorithms could handle were 64 and 4096, respectively. On the other two benchmarks, FORCLIFT has the lowest runtime, but, as it is finite-precision, only scales up to domain sizes of 16 and 128 in the case of 'friends' and 'functions', respectively. Next, FASTWFOMC outperforms FORCLIFT in the case of 'friends' but not 'functions': it scales up to domains of size 1024 and 64, respectively. Finally, both CRANE2-BFS and CRANE2-GREEDY perform nearly identically on both of these benchmarks. Similarly to the 'bijections' benchmark, CRANE2 significantly surpasses the other two algorithms, scaling up to domains of size 8192 and $67\,108\,864$, respectively.

Another aspect of the experimental results worth discussing separately is that of compilation. As both Julia and Scala use just-in-time (JIT) compilation, both FASTWFOMC and FORCLIFT take longer when run on the smallest domain size, as that is when most JIT compilation happens. As CRANE2 is run only once per benchmark, JIT compilation time is incorporated into its runtime across all domain sizes. Moreover, while FORCLIFT compilation is generally faster than that of CRANE2, neither is significant in comparison to inference time. Indeed, typically, FORCLIFT compilation took around $0.5\,\mathrm{s}$, while CRANE2 compilation took around $2.3\,\mathrm{s}$.

So what do our experiments reveal about which algorithm

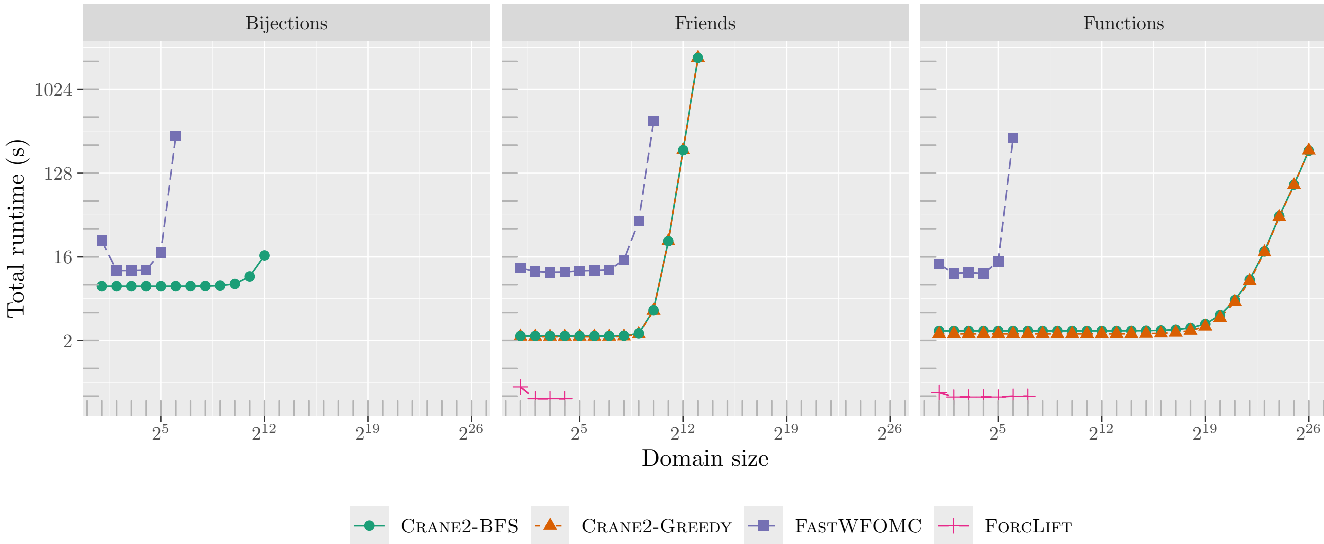

Figure 2: The runtime of the algorithms as a function of the domain size. Note that both axes are on a logarithmic scale.

should one use in practice? If the formula can be handled by FORCLIFT and the domain sizes are reasonably small, it is likely to be the fastest algorithm. In other cases, CRANE2 is likely to be significantly faster than FASTWFOMC across all domain sizes (but especially with higher domain sizes), assuming the formula can be handled by both algorithms.

7 Conclusion and Future Work

In this work, we have presented several contributions. First, we have developed algorithmic techniques to find the base cases of recursive functions generated by the original CRANE algorithm. Second, we have extended the smoothing procedure of FORCLIFT and CRANE to support base case formulas. Third, we have proposed an approach to compile function definitions into C++ programs with support for arbitrary-precision arithmetic. Lastly, we have provided experimental evidence demonstrating that CRANE2 can scale to much larger domain sizes than FASTWFOMC while handling more formulas than FORCLIFT.

There are many potential avenues for future work. Specifically, a more thorough experimental study is needed to understand how WFOMC algorithms compare in terms of their ability to handle different formulas and their scalability with respect to domain size. Additionally, further characterisation of the capabilities of CRANE2 can be explored. For example, *completeness* could be proven for a fragment of first-order logic such as C^2 (using a suitable encoding of counting quantifiers). Moreover, the efficiency of a WFOMC algorithm in handling a particular formula can be assessed using *fine-grained complexity*. In the case of CRANE and CRANE2, this can be done by analysing the equations (Dilkas and Belle 2023). By doing so, efficiency can be reasoned about in a more implementation-independent manner by making claims about the maximum degree of the polynomial that characterises any given solution.

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.

Hodges, W. 1997. *A Shorter Model Theory*. Cambridge University Press.

Jaeger, M.; and Van den Broeck, G. 2012. Liftability of Probabilistic Inference: Upper and Lower Bounds. In *StarAI@UAI*.

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.

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.

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.

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.

Russell, S.; and Norvig, P. 2020. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson.

Singla, P.; and Domingos, P. M. 2008. Lifted First-Order Belief Propagation. In *AAAI*, 1094–1099. AAAI Press.

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. 2011. On the Completeness of First-Order Knowledge Compilation for Lifted Probabilistic Inference. In *NIPS*, 1386–1394.

Van den Broeck, G.; Choi, A.; and Darwiche, A. 2012. Lifted Relax, Compensate and then Recover: From Approximate to Exact Lifted Probabilistic Inference. In *UAI*, 131–141. AUAI Press.

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.; 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.

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.; Pu, J.; Wang, Y.; and Kuželka, O. 2023. On Exact Sampling in the Two-Variable Fragment of First-Order Logic. In *LICS*, 1–13.

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.