Towards Practical First-Order Model Counting

Ananth K. Kidambi
Indian Institute of Technology Bombay, Mumbai, India

Guramrit Singh
Indian Institute of Technology Bombay, Mumbai, India

Paulius Dilkas
University of Toronto, Toronto, Canada
Vector Institute, Toronto, Canada

Kuldeep S. Meel
University of Toronto, Toronto, Canada

Abstract

First-order model counting (FOMC) is the problem of counting the number of models of a sentence in first-order logic. Since lifted inference techniques rely on reductions to variants of FOMC, the design of scalable methods for FOMC has attracted attention from both theoreticians and practitioners over the past decade. Recently, a new approach based on first-order knowledge compilation was proposed. This approach, called Crane, instead of simply providing the final count, generates definitions of (possibly recursive) functions that can be evaluated with different arguments to compute the model count for any domain size. However, this approach is not fully automated, as it requires manual evaluation of the constructed functions. The primary contribution of this work is a fully automated compilation algorithm, called Gantry, which 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.

2012 ACM Subject Classification Theory of computation → Automated reasoning; Theory of computation → Logic and verification; Mathematics of computing → Combinatorics

Keywords and phrases first-order model counting, knowledge compilation, lifted inference

Funding This research was funded in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), funding reference number RGPIN-2024-05956, and the Digital Research Alliance of Canada (`alliancecan.ca`).

Acknowledgements The first two authors contributed equally. Part of the research was conducted while all authors were at the National University of Singapore.

For the entire paper:
- 15 pages excluding references
- Make sure all the internal references (e.g., to examples) are still valid
- Maybe add extra references to subsections
- When submitting: go through the rest of formatting instructions (in GTD)

1 Introduction

First-order model counting (FOMC) is the task of determining the number of models for a sentence in first-order logic over a specified domain. The weighted variant, WFOMC, computes the total weight of these models, linking logical reasoning with probabilistic frameworks [31]. It builds upon earlier efforts in weighted model counting for propositional logic [4] and broader attempts to bridge logic and probability [14, 16, 20]. WFOMC is central to *lifted inference*, which enhances the efficiency of probabilistic calculations by exploiting symmetries [11]. Lifted inference continues to advance, with applications extending to constraint satisfaction problems [25] and probabilistic answer set programming [1]. Moreover, WFOMC has proven effective at reasoning over probabilistic databases [8] and probabilistic logic programs [18]. FOMC algorithms have also facilitated breakthroughs in discovering integer sequences [22] and developing recurrence relations for these sequences [6]. Recently, these algorithms have been extended to perform sampling tasks [32].

The complexity of FOMC is generally measured by *data complexity*, with a sentence classified as *liftable* if it can be solved in polynomial time relative to the domain size [10]. While all sentences with up to two variables are known to be liftable [28, 30], Beame et al. [3] demonstrated that liftability does not extend to all sentences, identifying an unliftable sentence with three variables. Recent work has further extended the liftable fragment with additional axioms [23, 27] and counting quantifiers [12], expanding our understanding of liftability.

FOMC algorithms are diverse, with approaches ranging from *first-order knowledge compilation* (FOKC) to cell enumeration [26], local search [15], and Monte Carlo sampling [7]. (See the technical appendix for a more detailed comparison of the state-of-the-art FOMC algorithms.) Among these, FOKC-based algorithms are particularly prominent, transforming sentences into structured representations such as circuits or graphs. Even when multiple algorithms are able to solve the same instance, FOKC algorithms are known to find polynomial-time solutions, where the polynomial has a lower degree compared to other approaches [6]. The recently developed ability of a FOKC algorithm to formulate solutions in terms of recursive functions [6] is also noteworthy as the only other proposed alternative is to guess recursive relations [2]. Notable examples of FOKC algorithms include ForcLift [31] and its successor Crane [6].

The Crane algorithm marked a significant step forward, expanding the range of sentences handled by FOMC algorithms. However, it had notable limitations: it required manual evaluation of function definitions to compute model counts and introduced recursive functions without proper base cases, making it more complex to use. To address these shortcomings, we present Gantry, a fully automated FOMC algorithm that overcomes the constraints of its predecessor. Gantry can handle domain sizes over 500,000 times larger than previous algorithms and simplifies the user experience by automatically handling base cases and compiling function definitions into efficient C++ programs.

The paper is structured as follows. Section 2 covers the necessary preliminaries including notation, terminology, and background information about (W)FOMC and FOKC. Section 3 describes all aspects of Gantry:

- its overall structure and interactions with Crane,
- how we identify a sufficient set of base cases for a recursive function,
- how we construct the sentence that ‘describes’ each base case, and
- how function definitions are compiled into a C++ program with caching mechanisms that ensure efficiency.

Then, Section 4 presents our experimental results, demonstrating Gantry’s performance compared to other FOMC algorithms. Finally, Section 5 concludes the paper by discussing promising avenues for future work.

2 Preliminaries

We begin this section by describing some notation that we will use throughout the paper. Then, in Sections 2.1 and 2.2 respectively, we introduce the basic terminology of first-order logic and formally define (W)FOMC. Section 2.3 outlines the principles of FOKC, particularly in the context of Crane. Finally, in Section 2.4, we introduce the algebraic terminology used to describe the output of Crane, i.e., functions and equations that define them.

Notation

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 $\sigma = [x_1 \mapsto y_1, x_2 \mapsto y_2, \dots, x_n \mapsto y_n]$ signifies the replacement of all instances of x_i with y_i for all $i = 1, \dots, n$. Additionally, for any variable n and $a, b \in \mathbb{N}_0$, let $[a \le n \le b] \coloneqq \begin{cases} 1 & \text{if } a \le n \le b \\ 0 & \text{otherwise} \end{cases}$.

2.1 First-Order Logic

In this section, we will review the basic concepts of first-order logic as they are used in FOKC algorithms. We begin by introducing the format used internally by ForcLift and its descendants. Afterwards, we provide a high-level description of how an arbitrary sentence in first-order logic is transformed into this internal format.

A *term* can be either a variable or a constant. An *atom* can be either $P(t_1, \dots, t_m)$ (i.e., $P(\mathbf{t})$) for some predicate P and terms $t_1, \dots, t_m$ or $x = y$ 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 the predicate P mentioned above. We write P/m to denote a predicate along with its arity. A *literal* can be 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 $\forall x_1 \in \Delta_1.\ \forall x_2 \in \Delta_2 \dots\ \forall x_n \in \Delta_n.\ \phi(x_1, x_2, \dots, x_n)$, where ϕ is a disjunction of literals that only contain variables $x_1, \dots, x_n$ (and any constants). We say that a clause is a *(positive) unit clause* if there is only one literal with a predicate, and it is a positive literal. Finally, a *sentence* is a conjunction of clauses. Throughout the paper, we will use set-theoretic notation, interpreting a sentence as a set of clauses and a clause as a set of literals.

▶ Remark. Conforming with previous work [31], the definition of a clause includes universal quantifiers for all variables within. While it is possible to rewrite the entire sentence with all quantifiers at the front [9], the format we describe has proven itself convenient to work with.

2.2 First-Order Model Counting

In this section, we will formally define FOMC and its weighted variant. Note that, although this work focuses on FOMC, for sentences with existential quantifiers, computing the FOMC using Gantry requires the use of WFOMC. For such sentences, preprocessing (described in Section 2.3) introduces predicates with non-unary weights that must be accounted for to compute the correct model count.

▶ **Definition 1** (Structure, model). *Let ϕ be a sentence in* FO. *For each predicate P/n in ϕ, let $(\Delta_i^P)_{i=1}^n$ be a list of the corresponding domains. Let σ be a map from the domains of ϕ to their interpretations as finite sets such that the sets are pairwise disjoint, and the constants in ϕ are included in the corresponding domains. A* structure *of ϕ 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 makes ϕ valid.*

▶ **Example 2** (Counting bijections). Let us consider the following sentence (previously examined by Dilkas and Belle [6]) that defines predicate P as a bijection between two domains Γ and Δ:

$$\begin{gathered}(\forall x \in \Gamma.\ \exists y \in \Delta.\ P(x,y)) \land \\ (\forall y \in \Delta.\ \exists x \in \Gamma.\ P(x,y)) \land \\ (\forall x \in \Gamma.\ \forall y, z \in \Delta.\ P(x,y) \land P(x,z) \Rightarrow y = z) \land \\ (\forall x, z \in \Gamma.\ \forall y \in \Delta.\ P(x,y) \land P(z,y) \Rightarrow x = z).\end{gathered} \tag{1}$$

Let σ be defined as $\sigma(\Gamma) := \{1,2\}$, and $\sigma(\Delta) := \{a,b\}$. Then Sentence (1) has two models:

$$\{P(1,a), P(2,b), \neg P(1,b), \neg P(2,a)\} \quad \text{and} \quad \{P(1,b), P(2,a), \neg P(1,a), \neg P(2,b)\}.$$

▶ Remark. The distinctness of domains is important in two ways. First, in terms of expressiveness, a clause such as $\forall x \in \Delta.\ P(x,x)$ is 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 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 3** (WFOMC instance). *A* WFOMC instance *comprises: a sentence ϕ in* FO*, two (rational)* weights *$w^+(P)$ and $w^-(P)$ assigned to each predicate P in ϕ, and σ as described in Definition 1. Unless specified otherwise, we assume all weights to be equal to 1.*

▶ **Definition 4** (WFOMC [31]). *Given a WFOMC instance (ϕ, w^+, w^-, σ) as in Definition 3, the* (symmetric) weighted first-order model count *(WFOMC) of ϕ is*

$$\sum_{M \models \phi} \prod_{P(\mathbf{t}) \in M} w^+(P) \prod_{\neg P(\mathbf{t}) \in M} w^-(P), \tag{2}$$

where the sum is over all models of ϕ.

2.3 Crane and First-Order Knowledge Compilation

As our work builds on Crane, in this section we will briefly outline the steps Crane goes through to compile an FO sentence into a set of function definitions. We divide the inner workings of the algorithm into two stages: preprocessing and compilation.

2.3.1 Preprocessing

The goal of this stage is to transform an arbitrary FO sentence into the format described in Section 2.1, most importantly by eliminating existential quantifiers. For example, the first conjunct of Sentence (1), i.e.,

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

is transformed into

$$\begin{gathered}(\forall x \in \Gamma.\ Z(x)) \land \\ (\forall x \in \Gamma.\ \forall y \in \Delta.\ Z(x) \lor \neg P(x,y)) \land \\ (\forall x \in \Gamma.\ S(x) \lor Z(x)) \land \\ (\forall x \in \Gamma.\ \forall y \in \Delta.\ S(x) \lor \neg P(x,y)),\end{gathered} \tag{4}$$

where $Z/1$ and $S/1$ are two new predicates with $w^-(S) = -1$. One can check that the WFOMC of Sentence (3) and (4) is the same.

2.3.2 Compilation

At this stage, the preprocessed sentence is compiled into the set $\mathcal{E}$ of equations and two auxiliary maps $\mathcal{F}$ and $\mathcal{D}$. $\mathcal{F}$ maps function names to sentences, and $\mathcal{D}$ maps function names and argument indices to domains. $\mathcal{E}$ can contain any number of functions, one of which (which we will always denote by f) represents the solution to the FOMC problem. To compute the FOMC for particular domain sizes, f must be evaluated with those domain sizes as arguments. $\mathcal{D}$ records this correspondence between function arguments and domains.

▶ **Example 5.** Crane compiles Sentence (1) for bijection counting into

$$\mathcal{E} = \left\{ \begin{aligned} f(m,n) &= \sum_{l=0}^{n} \binom{n}{l} (-1)^{n-l} g(l,m), \\ g(l,m) &= \sum_{k=0}^{m} [0 \le k \le 1] \binom{m}{k} g(l-1, m-k) \end{aligned} \right\};$$

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

where $\Delta^{\top}$ is a newly introduced domain. (We omit the definition of $\mathcal{F}$ as the sentences can get quite verbose.) To compute the number of bijections between two sets of cardinality 3, one would evaluate $f(3,3)$, however, the definition of g is incomplete: g is a recursive function presented without any base cases. $\mathcal{D}$ encodes that in $f(m,n)$, m and n represent $|\Gamma|$ and $|\Delta|$, respectively. Similarly, in $g(l,m)$, l represents $|\Delta^{\top}|$, and m represents $|\Gamma|$.

Compilation is performed primarily by applying *(compilation) rules* to sentences. Crane has two modes depending on how the algorithm chooses which compilation rule to apply to a sentence (in case several alternatives are available). The first option is to use greedy search: there is a list of rules, and the first applicable rule is the one that gets used, disregarding all the others. The second option is to use a combination of greedy and *breadth-first search* (BFS). That is, each compilation rule is identified as either greedy or non-greedy. Greedy rules are applied as soon as possible at any stage of the compilation process. BFS is executed over all applicable non-greedy rules, identifying the solution that can be constructed using the smallest number of such rules.

2.4 Algebra

In this paper, we use both logical and algebraic constructs. While the rest of Section 2 focused on the former, this section describes the latter. We write `expr` for an arbitrary algebraic expression. In the context of algebra, a *constant* is 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 call* is $f(x_1, \ldots, x_n)$ (or $f(\mathbf{x})$ for short), where f is an n-ary function, and each x_i is an algebraic expression consisting of variables and constants. A (function) *signature* is function call that contains only variables. Given two function calls $f(\mathbf{x})$ and $f(\mathbf{y})$, we say that $f(\mathbf{y})$ *matches* $f(\mathbf{x})$ if $x_i = y_i$ whenever $x_i, y_i \in \mathbb{N}_0$. An *equation* is $f(\mathbf{x}) =$ `expr`, where $f(\mathbf{x})$ is a function call.

▶ **Definition 6** (Base case). *Let $f(\mathbf{x})$ be a function call where each x_i is either a constant or a variable. Then function call $f(\mathbf{y})$ is 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.*

▶ **Example 7.** In equation $f(m,n) = f(m-1,n) + nf(m-1,n-1)$, the only constant is 1, and the variables are m and n. The equation contains three function calls: one on the

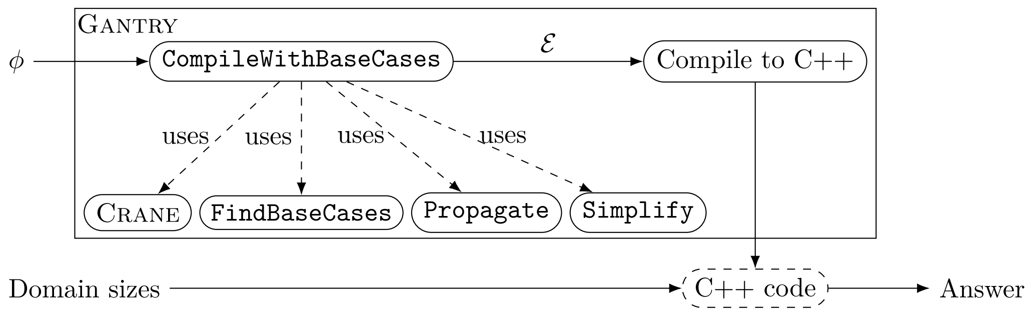

Figure 1 The outline of using Gantry to compute the model count of a sentence ϕ. First, ϕ is compiled into a set of equations, which are then used to create a C++ program. This program can be executed with different command line arguments to calculate the model count of ϕ for different domain sizes. To accomplish this, the `CompileWithBaseCases` procedure uses Crane, algebraic simplification techniques (denoted as `Simplify`), and two other auxiliary procedures.

left-hand side (LHS), and two on the right-hand side (RHS). The function call on the LHS is a signature. Function calls such as $f(4, n)$, $f(m, 0)$, and $f(8, 1)$ are all considered base cases of $f(m, n)$ (only some of which are useful).

3 Technical Contributions

Figure 1 provides an overview of Gantry's workflow. Below we briefly describe and motivate each procedure before going into more detail in the corresponding subsection.

`CompileWithBaseCases` (see Section 3.1), the core procedure of Gantry, is responsible for completing the function definitions produced by Crane with the necessary base cases. To do so, it may recursively call itself (and Crane) on other sentences. We prove that the number of such recursive calls is upper bounded by the number of domains in the sentences.

Section 3.1 also describes the `Simplify` procedure for algebraic simplification. It is crucial for simplifying, e.g., a sum of n terms, only two of which are non-zero. More generally, the equations returned by Crane often benefit from easy-to-detect algebraic simplifications such as $0 \cdot \text{anything} = 0$ and $\text{anything}^0 = 1$.

`FindBaseCases` (described in Section 3.2) inspects a set of equations to identify a sufficient set of base cases for a given set of equations. We prove that the returned set of base cases is sufficient, and the evaluation of the resulting function definitions will never get stuck in an infinite loop.

Section 3.3 introduces the `Propagate` procedure that takes a sentence ϕ, a domain Δ, and $n \in \mathbb{N}_0$. It returns ϕ transformed with the assumption that $|\Delta| = n$, introducing n new constants and removing all variables quantified over Δ. For example, when computing a base case such as $f(0, y)$, `Propagate` will significantly simplify ϕ with the assumption that the domain associated with the first parameter of f (i.e., $\mathcal{D}(f, 1)$) is empty. When run on this simplified sentence, `CompileWithBaseCases` will return the equations for the base case $f(0, y)$.

Section 3.4 describes a new kind of *smoothing* used to ensure that `Propagate` preserves the correct model count. Smoothing is a well-known technique in knowledge compilation algorithms for propositional model counting [5]. Although it has been applied to FOMC before [31], our setting requires a novel approach.

`CompileWithBaseCases`, together with the other procedures outlined above, return a set of equations that fully cover the base cases of all recursive functions. While these equations

Algorithm 1 `CompileWithBaseCases`(ϕ)

```
Input: sentence $\phi$
Output: set $\mathcal{E}$ of equations
1 $(\mathcal{E}, \mathcal{F}, \mathcal{D}) \leftarrow$ CRANE($\phi$);
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 $\psi \leftarrow$ Propagate($\psi$, $\mathcal{D}(f, i)$, $x_i$);
6     $\mathcal{E} \leftarrow \mathcal{E} \cup$ CompileWithBaseCases($\psi$);
```

can be interesting and valuable in their own right, the users of FOMC algorithms typically expect a numerical answer. Thus, Section 3.5 describes how these equations are compiled into a C++ program that can be executed with different command-line arguments to compute the model count for different combinations of domain sizes.

3.1 Completing the Definitions of Functions

Algorithm 1 presents our overall approach for compiling a sentence into equations that include the necessary base cases. To begin, we use CRANE to compile the sentence into the three components: $\mathcal{E}$, $\mathcal{F}$, and $\mathcal{D}$ (as described in Section 2.3.2). After some algebraic simplifications (described below), $\mathcal{E}$ is passed to the `FindBaseCases` procedure (see Section 3.2). For each base case $f(\mathbf{x})$, we retrieve the sentence $\mathcal{F}(f)$ associated with the function name f and simplify it using the `Propagate` procedure (explained in detail in Section 3.3). 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 6, `CompileWithBaseCases` recurses on these simplified sentences and adds the resulting base case equations to $\mathcal{E}$.

Simplify

The main responsibility of the `Simplify` procedure is to handle the algebraic pattern $\sum_{m=0}^{n}[a \le m \le b]f(m)$. Here: n is a variable, $a, b \in \mathbb{N}_0$ are constants, and f is an expression that may depend on m. `Simplify` transforms this pattern into $f(a) + f(a+1) + \cdots + f(\min\{\, n, b\,\})$.

▶ **Example 8.** Let us return to the bijection-counting problem from Example 2 and its initial solution described in Example 5. `Simplify` transforms $g(l, m) = \sum_{k=0}^{m}[0 \le k \le 1]\binom{m}{k}g(l-1, m-k)$ into $g(l, m) = g(l-1, m) + mg(l-1, m-1)$. Then `FindBaseCases` identifies two base cases: $g(0, m)$ and $g(l, 0)$. In both cases, `CompileWithBaseCases` recurses on the sentence $\mathcal{F}(g)$ simplified by assuming that one of the domains is empty. In the first case, we recurse on the sentence $\forall x \in \Gamma.\ S(x) \lor \neg S(x)$, where S is a predicate introduced by preprocessing 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 sentence, resulting in $g(l, 0) = 1$. While these base cases overlap when $l = m = 0$, they remain consistent since $0^0 = 1$.

We end this section by proving that `CompileWithBaseCases` terminates since each recursive call on line 6 reduces the number of domains in the sentence.

▶ **Theorem 9.** *Given any* FO *sentence* ϕ, `CompileWithBaseCases`(ϕ) *terminates.*

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

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

Reformulate the theorem above into an upper bound (as I informally mention earlier in the text)

To prove the theorem, we rely on two observations about the algorithms presented in Sections 3.2 and 3.3.

▶ **Observation 10.** *Each base case returned by* `FindBaseCases` *has at least one constant (in line with Definition 6).*

▶ **Observation 11.** *For any sentence ϕ, domain Δ, and $n \in \mathbb{N}_0$,* `Propagate(`ϕ, Δ, n`)` *returns a sentence with no variables quantified over Δ.*

Proof. We proceed by induction on the number of domains that variables in ϕ are quantified over. If there are no domains, then ϕ is essentially a propositional formula, and CRANE compiles it into an equation of the form $f = \texttt{expr}$ with no 'function calls'. Suppose that `CompileWithBaseCases` terminates for all sentences with at most $n \in \mathbb{N}_0$ domains. Let ϕ be a sentence with $n + 1$ domains. By Observation 10, each base case on line 3 of Algorithm 1 has at least one constant. Therefore, by Observation 11, after line 5, sentence ψ has at most n domains. Thus, line 6 terminates by the inductive hypothesis, completing the proof that `CompileWithBaseCases` terminates for an arbitrary sentence with $n + 1$ domains. ◀

3.2 Identifying a Sufficient Set of Base Cases

Algorithm 2 summarises the implementation of `FindBaseCases`. It considers two types of arguments when a function f calls itself recursively: constants and arguments of the form $x_i - c_i$. Here, c_i is a constant, and x_i is the i-th argument of the signature of f. When 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 0 up to (but not including) c_i.

▶ **Example 12.** Consider the recursive function g from Example 5. `FindBaseCases` 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 $g(l, 0)$.

It can be shown that the base cases identified by `FindBaseCases` are sufficient for the algorithm to terminate.[1] For the remainder of this section, let $\mathcal{E}$ denote the equations

[1] Note that characterising the fine-grained complexity of the solutions found by GANTRY or other FOMC algorithms is an emerging area of research. These questions have been partially addressed in previous work [6, 24] and are orthogonal to the goals of this section.

returned by `CompileWithBaseCases`.

▶ **Theorem 13.** *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 13 using double induction. First, we apply induction to 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. We begin with a few observations that stem from previous [6, 31] and this work.

▶ **Observation 14.** *For each function f, there is precisely one equation $e \in \mathcal{E}$ with $f(\mathbf{x})$ on the LHS 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 15.** *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 LHS do not contain function calls to f_j with $j > i$. This condition prevents mutual recursion and other cyclic scenarios.*

▶ **Observation 16.** *For each 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 17.** *If f is a non-recursive function with no function calls on the RHS of its definition, then the evaluation of any function call $f(\mathbf{x})$ terminates.*

▶ **Observation 18.** *For each equation $(f(\mathbf{x}) = \texttt{expr}) \in \mathcal{E}$, 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 19.** *For each 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 20.** *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 LHS.*

With the observations and assumptions mentioned above, we are ready to prove Theorem 13. For readability, we divide the proof into several lemmas of increasing generality.

▶ **Lemma 21.** *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 17 and Observation 18. If f is not recursive, the evaluation of $f(x)$ terminates by Corollary 17.

Otherwise, let $f(y)$ be an arbitrary function call on the RHS 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 $0 \le x - kc \le c - 1$. So, after k iterations, the sequence of function calls $f(x), f(x-c), f(x-2c), \ldots$ will be captured by the base case $f(x \mod c)$. ◀

▶ **Lemma 22.** *Generalising Lemma 21, let $\mathcal{E}$ be a set of equations for* one *n-ary function f for some $n \ge 1$. Then the evaluation of $f(\mathbf{x})$ terminates for any $\mathbf{x} \in \mathbb{N}_0^n$.*

Algorithm 3 `Propagate(`ϕ, Δ, n`)`

```
Input: sentence $\phi$, domain $\Delta$, $n \in \mathbb{N}_0$
Output: sentence $\phi'$
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 $\Delta$;
12     foreach clause $C \in \phi$ do
13         $(x_i)_{i=1}^m \leftarrow$ the variables in $C$ with domain $\Delta$;
14         if $m = 0$ then $\phi' \leftarrow \phi' \cup \{ C \}$;
15         else $\phi' \leftarrow \phi' \cup \{ C[x_1 \mapsto c_1, \ldots, x_m \mapsto c_m] \mid (c_i)_{i=1}^m \in D^m \}$;
```

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 21. 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 16. ◀

Proof of Theorem 13. We proceed by induction on the number of functions n. The base case of $n = 1$ is handled by Lemma 22. 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 15. Using the inductive hypothesis that all function calls to f_j (with $j < n$) terminate, the proof proceeds similarly to the Proof of Lemma 22. ◀

3.3 Propagating Domain Size Assumptions

Algorithm 3, called `Propagate`, modifies the sentence 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 sentences examined in this work had $n > 1$.) Algorithm 3 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:

1. those that do not mention Δ,
2. those in which every literal contains variables quantified over Δ, and
3. those that have some literals with variables quantified over Δ and some without.

Clauses of Type 1 are transferred to the new sentence ϕ' without any changes. For clauses of Type 2, C' is empty, so these clauses are filtered out. As for clauses of Type 3, a new kind of smoothing is performed, which will be explained in Section 3.4.

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

$$
\begin{aligned}
&(\forall x \in \Gamma.\ \neg P(x, c_1) \vee \neg P(x, c_1) \vee c_1 = c_1) \wedge \\
&(\forall x \in \Gamma.\ \neg P(x, c_1) \vee \neg P(x, c_2) \vee c_1 = c_2) \wedge \\
&\qquad\qquad\qquad\qquad \vdots \\
&(\forall x \in \Gamma.\ \neg P(x, c_3) \vee \neg P(x, c_3) \vee c_3 = c_3).
\end{aligned}
$$

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

3.4 Smoothing the Base Cases

Smoothing modifies a circuit to reintroduce eliminated atoms, ensuring the correct model count [5, 31]. In this section, we describe a similar process performed on lines 7–9 of Algorithm 3. Line 7 checks if 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 omitted.

Suppose `Propagate` is called with arguments $(\phi, \Delta, 0)$, i.e., we are simplifying the sentence 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 24.** Let ϕ be

$$
\begin{aligned}
&(\forall x \in \Delta.\ \forall y, z \in \Gamma.\ Q(x) \vee P(y, z)) \wedge && (5)\\
&\qquad\qquad (\forall y, z \in \Gamma'.\ P(y, z)), && (6)
\end{aligned}
$$

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 (6), but it can vary freely over $(\Gamma \times \Gamma) \setminus (\Gamma' \times \Gamma')$ since Clause (5) is vacuously satisfied by all structures. Therefore, the correct FOMC 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 sentence has only one model: $\{\, P(y, z) \mid y, z \in \Gamma' \,\}$. By including line 9, `Propagate` transforms ϕ to

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

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'.

Algorithm 4 A sketch of the C++ program for the equations in Example 5, particularly highlighting the recursive definition of function g.

```
1  initialise $\mathsf{Cache}_{g(0,m)}$, $\mathsf{Cache}_{g(l,0)}$, $\mathsf{Cache}_g$, and $\mathsf{Cache}_f$;
2  **Function** $g_{0,m}(m)$: ...
3  **Function** $g_{l,0}(l)$: ...
4  **Function** $g(l,m)$:
5      **if** $(l,m) \in \mathsf{Cache}_g$ **then return** $\mathsf{Cache}_g(l,m)$;
6      **if** $l = 0$ **then return** $g_{0,m}(m)$;
7      **if** $m = 0$ **then return** $g_{l,0}(l)$;
8      $r \leftarrow g(l-1,m) + mg(l-1,m-1)$;
9      $\mathsf{Cache}_g(l,m) \leftarrow r$;
10     **return** $r$;
11 **Function** $f(m,n)$: ...
12 **Function** Main:
13     $(m,n) \leftarrow$ ParseCommandLineArguments();
14     **return** $f(m,n)$;
```

3.5 Generating C++ Code

In this section, we will describe the final step of GANTRY as outlined in Figure 1, i.e., translating the set of equations $\mathcal{E}$ into C++ code. Recall that this step is crucial for the usability of the algorithm, otherwise function definitions would remain purely mathematical, with no convenient way to compute the model count for particular domain sizes. Once a C++ program is produced, it can be executed with different command-line arguments to compute the model count of the sentence for various domain sizes.

See Algorithm 4 for the typical structure of a generated C++ program. Each equation in $\mathcal{E}$ is compiled into a C++ function, along with a separate cache for memoisation. Hence, Algorithm 4 has a function and a cache for $f(\cdot,\cdot)$, $g(\cdot,\cdot)$, $g(\cdot,0)$, and $g(0,\cdot)$. The implementation of an equation consists of three parts. First (on line 5), we check if the arguments are already present in the corresponding cache. If so, we simply return the cached value. Second (on lines 6 and 7), for each base case, we check if the arguments match the base case (as defined in Section 2.4). If so, the arguments are redirected to the C++ function for that base case. Finally, if none of the above cases apply, we evaluate the arguments based on the expression on the RHS of the equation, store the result in the cache, and return it.

4 Experimental Evaluation

Our empirical evaluation sought to compare the runtime performance of GANTRY with the current state of the art, namely FASTWFOMC and FORCLIFT. Our experiments involve two versions of GANTRY: GANTRY-GREEDY and GANTRY-BFS. Like its predecessor (see Section 2.3.2), GANTRY has two modes for applying compilation rules to sentences: one that uses a greedy search algorithm similar to FORCLIFT and another that combines greedy and BFS.

The experiments were conducted using an Intel Skylake 2.4 GHz CPU with 188 GiB of memory and CentOS 7. C++ programs were compiled using the Intel C++ Compiler 2020u4. FASTWFOMC ran on Julia 1.10.4, while the other algorithms were executed on the Java Virtual Machine 1.8.0_201. Note that, although implemented in different languages, both Gantry and FastWFOMC use the GNU Multiple Precision Arithmetic Library for arbitrary-precision arithmetic.

We ran each algorithm on each benchmark using domains of sizes $2^1, 2^2, 2^3$, and so on, until an algorithm failed to handle a domain size due to timeout (of 1 h), out of memory error, or out of precision errors. While we separately measured compilation and inference time, we primarily focus on total runtime, dominated by the latter. We verified the accuracy of the numerical answers using the corresponding integer sequences on the On-Line Encyclopedia of Integer Sequences [17].

4.1 Benchmarks

We compare these algorithms using three benchmarks from previous work. The first benchmark is the bijection-counting problem from Example 2. The next benchmark is a variant of the well-known *Friends & Smokers* Markov logic network [21, 29], which can be formulated as

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

In this sentence, we have three predicates S, F, and C that denote smoking, friendship, and cancer, respectively. The first clause states that friends of smokers are also smokers, and the second clause asserts that smoking causes cancer. Common additions to this sentence include making the friendship relation symmetric and assigning probabilities to each clause. Finally, we include the function-counting problem [6]

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

as our last benchmark. Here, predicate P is defined as a function from Γ to Δ. The first clause asserts that each x must have at least one corresponding y, while the second clause ensures that there is only one such y.

▶ Remark. We formulate *Bijections* and *Functions* benchmarks using two domains Γ and Δ as such a formulation is known to help FOKC algorithms find efficient solutions [6]. To compare Gantry and ForcLift with FastWFOMC that has no support for multiple domains, we set $|\Gamma| = |\Delta|$.

The three benchmarks cover a wide range of possibilities. The *Friends & Smokers* benchmark uses multiple predicates and can be expressed in FO using two variables without cardinality constraints or counting quantifiers. The *Functions* benchmark, on the other hand, can still be handled by all the algorithms, but it requires cardinality constraints, counting quantifiers, or more than two variables, depending on the formulation and the capabilities of the algorithm. Lastly, the *Bijections* benchmark is an example of a sentence that FastWFOMC can handle but ForcLift cannot.

4.2 Results

Figure 2 presents a summary of the experimental results. Only FastWFOMC and Gantry-BFS could handle the bijection-counting problem. For this benchmark, the largest domain sizes these algorithms could accommodate were 64 and 4096, respectively. On the other two benchmarks, ForcLift had the lowest runtime. However, since it can only handle model counts smaller than 2^{31}, it only scales up to domain sizes of 16 and 128 for *Friends & Smokers* and *Functions*, respectively. FastWFOMC outperformed ForcLift in the case of *Friends & Smokers*, but not *Functions*, as it could handle domains of size 1024

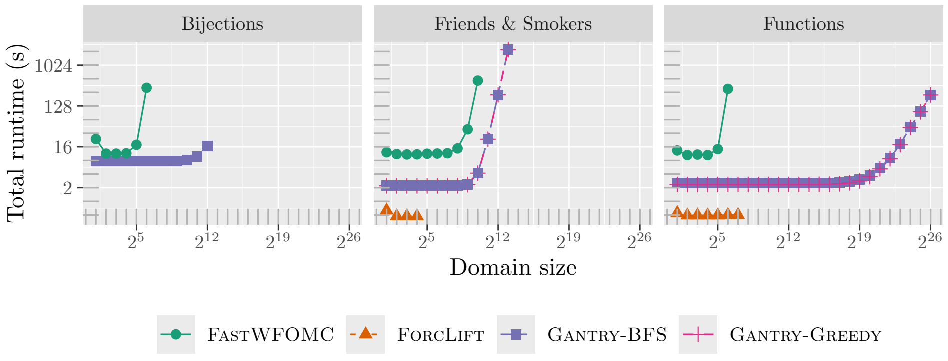

Figure 2 The runtime of the algorithms as a function of the domain size. Note that both axes are on a logarithmic scale.

and 64, respectively. Furthermore, both Gantry-BFS and Gantry-Greedy performed similarly on both benchmarks. Similarly to the *Bijections* benchmark, Gantry significantly outperformed the other two algorithms, scaling up to domains of size 8192 and 67,108,864, respectively.

One might notice that the runtime of FastWFOMC and ForcLift is slightly higher on the smallest domain size. This peculiarity is the consequence of *just-in-time* (JIT) compilation. As Gantry is only run once per benchmark, the JIT compilation time is included in its overall runtime across all domain sizes. Additionally, while ForcLift's compilation is generally faster than that of Gantry, neither significantly affects overall runtime. Specifically, ForcLift compilation typically takes around 0.5 s, while Gantry compilation takes around 2.3 s.

Based on our experiments, which algorithm should be used in practice? If the sentence can be handled by ForcLift and the domain sizes are reasonably small, ForcLift is likely the fastest algorithm. In other situations, Gantry is expected to be significantly more efficient than FastWFOMC regardless of domain size, provided both algorithms can handle the sentence.

5 Conclusion and Future Work

In this work, we have presented a scalable automated FOKC-based approach to FOMC. Our algorithm involves completing the definitions of recursive functions and subsequently translating all function definitions into C++ code. Empirical results demonstrate that Gantry can scale to larger domain sizes than FastWFOMC while supporting a wider range of sentences than ForcLift. The ability to efficiently handle large domain sizes is particularly crucial in the weighted setting, as illustrated by the *Friends & Smokers* example discussed in Section 4, where the model captures complex social networks with probabilistic relationships. Without this scalability, the practical usefulness of these models would be limited.

Future directions for research include conducting a comprehensive experimental comparison of FOMC algorithms to better understand their comparative performance across various sentences. The capabilities of Gantry could also be characterised theoretically, e.g., by proving completeness for logic fragments liftable by other algorithms. Additionally, the efficiency of FOMC algorithms can be further analysed using fine-grained complexity, which would provide more detailed insights into the computational demands of different sentences.

Cite a bunch of papers for the liftable fragments.

References

1 Damiano Azzolini and Fabrizio Riguzzi. Lifted inference for statistical statements in probabilistic answer set programming. *Int. J. Approx. Reason.*, 163:109040, 2023. doi:10.1016/J.IJAR.2023.109040.

2 Jáchym Barvínek, Timothy van Bremen, Yuyi Wang, Filip Zelezný, and Ondřej Kuželka. Automatic conjecturing of P-recursions using lifted inference. In *ILP*, volume 13191 of *Lecture Notes in Computer Science*, pages 17–25. Springer, 2021. doi:10.1007/978-3-030-97454-1_2.

3 Paul Beame, Guy Van den Broeck, Eric Gribkoff, and Dan Suciu. Symmetric weighted first-order model counting. In *PODS*, pages 313–328. ACM, 2015. doi:10.1145/2745754.2745760.

4 Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artif. Intell.*, 172(6-7):772–799, 2008. doi:10.1016/J.ARTINT.2007.11.002.

5 Adnan Darwiche. 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, 2001. doi:10.3166/JANCL.11.11-34.

6 Paulius Dilkas and Vaishak Belle. Synthesising recursive functions for first-order model counting: Challenges, progress, and conjectures. In *KR*, pages 198–207, 2023. doi:10.24963/KR.2023/20.

7 Vibhav Gogate and Pedro M. Domingos. Probabilistic theorem proving. *Commun. ACM*, 59(7):107–115, 2016. doi:10.1145/2936726.

8 Eric Gribkoff, Dan Suciu, and Guy Van den Broeck. Lifted probabilistic inference: A guide for the database researcher. *IEEE Data Eng. Bull.*, 37(3):6–17, 2014. URL: http://sites.computer.org/debull/A14sept/p6.pdf.

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

10 Manfred Jaeger and Guy Van den Broeck. Liftability of probabilistic inference: Upper and lower bounds. In *StarAI@UAI*, 2012. URL: https://starai.cs.kuleuven.be/2012/accepted/jaeger.pdf.

11 Kristian Kersting. Lifted probabilistic inference. In *ECAI*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, pages 33–38. IOS Press, 2012. doi:10.3233/978-1-61499-098-7-33.

12 Ondřej Kuželka. Weighted first-order model counting in the two-variable fragment with counting quantifiers. *J. Artif. Intell. Res.*, 70:1281–1307, 2021. doi:10.1613/JAIR.1.12320.

13 Sagar Malhotra and Luciano Serafini. Weighted model counting in FO2 with cardinality constraints and counting quantifiers: A closed form formula. In *AAAI*, pages 5817–5824. AAAI Press, 2022. doi:10.1609/AAAI.V36I5.20525.

14 Nils J. Nilsson. Probabilistic logic. *Artif. Intell.*, 28(1):71–87, 1986. doi:10.1016/0004-3702(86)90031-7.

15 Feng Niu, Christopher Ré, AnHai Doan, and Jude W. Shavlik. Tuffy: Scaling up statistical inference in Markov logic networks using an RDBMS. *Proc. VLDB Endow.*, 4(6):373–384, 2011. doi:10.14778/1978665.1978669.

16 Vilém Novák, Irina Perfilieva, and Jiri Mockor. *Mathematical principles of fuzzy logic*, volume 517. Springer Science & Business Media, 2012.

17 OEIS Foundation Inc. The On-Line Encyclopedia of Integer Sequences, 2025. Published electronically at http://oeis.org.

18 Fabrizio Riguzzi, Elena Bellodi, Riccardo Zese, Giuseppe Cota, and Evelina Lamma. A survey of lifted inference approaches for probabilistic logic programming under the distribution semantics. *Int. J. Approx. Reason.*, 80:313–333, 2017. doi:10.1016/J.IJAR.2016.10.002.

19 Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020.

20 Dragan Z. Šaletić. Graded logics. *Interdisciplinary Description of Complex Systems: INDECS*, 22(3):276–295, 2024.

21 Parag Singla and Pedro M. Domingos. Lifted first-order belief propagation. In *AAAI*, pages 1094–1099. AAAI Press, 2008. URL: `http://www.aaai.org/Library/AAAI/2008/aaai08-173.php`.

22 Martin Svatos, Peter Jung, Jan Tóth, Yuyi Wang, and Ondřej Kuželka. On discovering interesting combinatorial integer sequences. In *IJCAI*, pages 3338–3346. ijcai.org, 2023. `doi:10.24963/IJCAI.2023/372`.

23 Jan Tóth and Ondřej Kuželka. Lifted inference with linear order axiom. In *AAAI*, pages 12295–12304. AAAI Press, 2023. `doi:10.1609/AAAI.V37I10.26449`.

24 Jan Tóth and Ondřej Kuželka. Complexity of weighted first-order model counting in the two-variable fragment with counting quantifiers: A bound to beat. In *KR*, 2024. `doi:10.24963/KR.2024/64`.

25 Pietro Totis, Jesse Davis, Luc De Raedt, and Angelika Kimmig. Lifted reasoning for combinatorial counting. *J. Artif. Intell. Res.*, 76:1–58, 2023. `doi:10.1613/JAIR.1.14062`.

26 Timothy van Bremen and Ondřej Kuželka. Faster lifting for two-variable logic using cell graphs. In *UAI*, volume 161 of *Proceedings of Machine Learning Research*, pages 1393–1402. AUAI Press, 2021. URL: `https://proceedings.mlr.press/v161/bremen21a.html`.

27 Timothy van Bremen and Ondřej Kuželka. Lifted inference with tree axioms. *Artif. Intell.*, 324:103997, 2023. `doi:10.1016/J.ARTINT.2023.103997`.

28 Guy Van den Broeck. On the completeness of first-order knowledge compilation for lifted probabilistic inference. In *NIPS*, pages 1386–1394, 2011. URL: `https://proceedings.neurips.cc/paper/2011/hash/846c260d715e5b854ffad5f70a516c88-Abstract.html`.

29 Guy Van den Broeck, Arthur Choi, and Adnan Darwiche. Lifted relax, compensate and then recover: From approximate to exact lifted probabilistic inference. In *UAI*, pages 131–141. AUAI Press, 2012. URL: `https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=2349&proceeding_id=28`.

30 Guy Van den Broeck, Wannes Meert, and Adnan Darwiche. Skolemization for weighted first-order model counting. In *KR*. AAAI Press, 2014. URL: `http://www.aaai.org/ocs/index.php/KR/KR14/paper/view/8012`.

31 Guy Van den Broeck, Nima Taghipour, Wannes Meert, Jesse Davis, and Luc De Raedt. Lifted probabilistic inference by first-order knowledge compilation. In *IJCAI*, pages 2178–2185. IJCAI/AAAI, 2011. `doi:10.5591/978-1-57735-516-8/IJCAI11-363`.

32 Yuanhong Wang, Juhua Pu, Yuyi Wang, and Ondřej Kuželka. Lifted algorithms for symmetric weighted first-order model sampling. *Artif. Intell.*, 331:104114, 2024. `doi:10.1016/J.ARTINT.2024.104114`.