Recursive Solutions to First-Order Model Counting

Anonymous submission

Abstract

First-order model counting (FOMC) is a computational problem that asks to count the models of a sentence in finite-domain first-order logic. Despite being around for more than a decade, practical FOMC algorithms are still unable to compute functions as simple as a factorial. We argue that the capabilities of FOMC algorithms to date are limited by their inability to express arbitrary recursive computations. To enable arbitrary recursion, we relax the restrictions that typically accompany domain recursion and generalise circuits used to express a solution to an FOMC problem to graphs that may contain cycles. To this end, we enhance the most well-established (weighted) FOMC algorithm FORCLIFT with new compilation rules and an algorithm to check whether a recursive call is feasible. These improvements allow the algorithm to construct efficient solutions to counting fundamental structures such as injections and bijections. We end with a few conjectures on what classes of instances could be liftable as a result.

1 Introduction

First-order model counting (FOMC) is the problem of computing the number of models of a sentence in first-order logic (FOL) given the size(s) of its domain(s) (Beame et al. 2015). *Symmetric weighted FOMC* (WFOMC) extends FOMC with (pairs of) weights on predicates and asks for a weighted sum across all models instead. By fixing the sizes of the domains, a WFOMC instance can be rewritten as an instance of (propositional) weighted model counting (Chavira and Darwiche 2008). WFOMC emerged as the dominant approach to *lifted (probabilistic) inference*. Lifted inference techniques exploit symmetries in probabilistic models by reasoning about sets rather than individuals (Kersting 2012). By doing so, many instances become solvable in polynomial time (Van den Broeck 2011). Lifted inference algorithms are typically used on probabilistic models such as probabilistic programming languages (De Raedt and Kimmig 2015; Riguzzi et al. 2017), Markov logic networks (Van den Broeck et al. 2011; Gogate and Domingos 2016; Richardson and Domingos 2006), and other lifted graphical (Kimmig, Mihalkova, and Getoor 2015) and statistical relational (De Raedt et al. 2016) models. Lifted inference techniques for probabilistic databases, while developed somewhat independently, have also been inspired by WFOMC (Gatterbauer and Suciu 2015; Gribkoff, Suciu, and Van den Broeck 2014). While WFOMC tends to receive more attention in the literature, FOMC is an interesting problem in an of itself because of its connections to finite model theory (van Bremen and Kuzelka 2021b) and applications in enumerative combinatorics (Barvínek et al. 2021).

Traditionally in computational complexity theory, a problem is *tractable* if it can be solved in time polynomial in the instance size. The equivalent notion in (W)FOMC is liftability. A (W)FOMC instance is *(domain-)liftable* if it can be solved in time polynomial in the size(s) of the domain(s) (Jaeger and Van den Broeck 2012). Over more than a decade, many classes of instances were shown to be liftable (van Bremen and Kuzelka 2021b; Kazemi et al. 2016; Kuusisto and Lutz 2018; Kuzelka 2021). First, Van den Broeck (2011) showed that the class of all sentences of FOL with up to two variables (denoted FO^2) is liftable. Then Beame et al. (2015) proved that there exists a sentence with three variables for which FOMC is $\#\mathsf{P}_1$-complete (i.e., FO^3 is not liftable). Since these two results came out, most of the research on (W)FOMC focused on developing faster solutions for the FO^2 fragment (van Bremen and Kuzelka 2021a; Malhotra and Serafini 2022) and defining new liftable fragments. These fragments include $\mathsf{S}^2\mathsf{FO}^2$ and $\mathsf{S}^2\mathsf{RU}$ (Kazemi et al. 2016), U_1 (Kuusisto and Lutz 2018), FO^2 with tree axioms (van Bremen and Kuzelka 2021b), and C^2 (i.e., the two-variable fragment with counting quantifiers) (Kuzelka 2021; Malhotra and Serafini 2022). On the empirical front, there are several open-source implementations of exact WFOMC algorithms: FORCLIFT (Van den Broeck et al. 2011), probabilistic theorem proving (Gogate and Domingos 2016), and L2C (Kazemi and Poole 2016). Approximate counting is supported by APPROXWFOMC (van Bremen and Kuzelka 2020) as well as FORCLIFT (Van den Broeck, Choi, and Darwiche 2012) and probabilistic theorem proving (Gogate and Domingos 2016).

However, none of the publicly available exact (W)FOMC algorithms can efficiently compute functions as simple as a factorial.[1] We claim that this shortcoming is due to the inability of these algorithms to construct recursive solutions.

[1] The problem of computing the factorial can be described using two variables and counting quantifiers, so it is known to be liftable in principle (Kuzelka 2021), but an algorithm that can lift such an instance in practice does not exist.

The topic of recursion in the context of WFOMC has been studied before but in limited ways. Barvínek et al. (2021) use WFOMC to generate numerical data that is then used to conjecture recurrence relations that explain that data. Van den Broeck (2011) introduced the idea of *domain recursion*. Intuitively, domain recursion partitions a domain of size n into a single explicitly named constant and the remaining domain of size $n-1$. However, many stringent conditions are enforced to ensure that the search for a tractable solution always terminates.

In this work, we show how to relax these restrictions in a way that results in a stronger (W)FOMC algorithm, capable of handling more instances (e.g., counting various injective mappings) in a lifted manner. The ideas presented in this paper are implemented in CRANE—an extension of the arguably most well-known WFOMC algorithm FORCLIFT. FORCLIFT works in two stages: compilation and evaluation/propagation. In the first stage, various *(compilation) rules* are applied to the input (or some derivative) formula, gradually constructing a circuit. In the second stage, the weights of the instance are propagated through the circuit, computing the weighted model count. Along with new compilation rules, CRANE introduces changes to both stages of the process. First, while FORCLIFT applies compilation rules via greedy[2] search, CRANE uses a hybrid search algorithm that applies some rules greedily and some using breadth-first search. Second, the product of compilation is not directly evaluated but rather interpreted as a collection of functions on domain sizes.

The main conceptual difference between CRANE and FORCLIFT is that we utilise labelled directed graphs instead of circuits, where cycles represent recursive calls. Suppose the input formula ϕ depends on a domain of size $n \in \mathbb{N}$. *Generalised domain recursion* (GDR)—one of the new compilation rules—transforms ϕ into a different formula ψ that has an additional constant and some new *constraints*. After some additional transformations, the constraints in ψ become 'uniform' and can be removed, replacing the domain of size n with a new domain of size $n-1$—this is the responsibility of the *constraint removal* (CR) compilation rule. Afterwards, another compilation rule recognizes that the resulting formula is just like the input formula ϕ but with a different domain. This observation allows us to add a cycle-forming edge to the graph, which can be interpreted as function f relying on $f(n-1)$ to compute $f(n)$.

We begin by defining the representation used for sentences in FOL in Section 2. Then, in Section 3, we define the graphs that replace circuits in representing a solution to a (W)FOMC problem. Section 4 introduces the new compilation rules. In Section 5, we discuss how a graph can be interpreted as a collection of (potentially recursive) functions. In Section 6, we compare FORCLIFT and CRANE on a range of function-counting problems. We show that CRANE performs as well as FORCLIFT on the instances that were already solvable by FORCLIFT but is also able to handle most of the instances that FORCLIFT fails on. Finally, in Section 7, we conclude by outlining some conjectures and directions for future work.

[2]The algorithm is not described in any paper but can be found in its source code at https://dtai.cs.kuleuven.be/drupal/wfomc

2 Preliminaries

Our representation of FOMC instances is largely based on the format used internally by FORCLIFT, some aspects of which are described by Van den Broeck et al. (2011). FORCLIFT can translate sentences in a variant of function-free many-sorted FOL with equality to this internal format. We use lowercase Latin letters for predicates and constants, uppercase Latin letters for variables, and uppercase Greek letters for domains. An *atom* is $p(t_1, \ldots, t_n)$ for some predicate p and terms $t_1, \ldots, t_n$. A *term* is either a constant or a variable. A *literal* is either an atom or the negation of an atom (denoted by $\neg p(t_1, \ldots, t_n)$). Let $\mathcal{D}$ be the set of all (finite) domains. Initially, $\mathcal{D}$ contains all domains mentioned by the input (W)FOMC instance. During compilation, new domains are added to $\mathcal{D}$ by some of the compilation rules. Each such new domain is interpreted as a subset of some element of $\mathcal{D}$.

Definition 1 (Constraint)**.** An *(inequality) constraint* is a pair (a, b), where a is a variable, and b is either a variable or a constant.

Definition 2 (Clause)**.** A *clause*[3] is a triple $c = (L, C, \delta_c)$, where L is a set of literals, C is a set of constraints, and δ_c is the domain map of c. Let Vars be the function that maps clauses and sets of either literals or constraints to the set of variables contained within. In particular, $\mathrm{Vars}(c) \coloneqq \mathrm{Vars}(L) \cup \mathrm{Vars}(C)$. *Domain map* $\delta_c \colon \mathrm{Vars}(c) \to \mathcal{D}$ is a function that maps all variables in c to their domains such that (s.t.) if $(X, Y) \in C$ for some variables X and Y, then $\delta_c(X) = \delta_c(Y)$. For convenience, we sometimes write δ_c for the domain map of c without unpacking c into its three constituents.

Similarly to variables in Definition 2, all constants are (implicitly) mapped to domains, and each n-ary predicate is implicitly mapped to a sequence of n domains. For constant or variable x, predicate p, and domains Γ and Δ, we write, e.g., $x \in \Gamma$ and $p \in \Gamma \times \Delta$ to denote that x is associated with Γ, and p is associated with Γ and Δ (in that order).

Definition 3 (Formula)**.** A *formula* (called a c-theory by Van den Broeck et al. (2011)) is a set of clauses s.t. all constraints and atoms 'type check' with respect to domains.

Example 1. Let $\phi \coloneqq \{\, c_1, c_2 \,\}$ be a formula with clauses

$$c_1 \coloneqq (\{\, \neg p(X, Y), \neg p(X, Z) \,\}, \{\, (Y, Z) \,\}, \{\, X \mapsto \Gamma, Y \mapsto \Delta, Z \mapsto \Delta \,\}),$$
$$c_2 \coloneqq (\{\, \neg p(X, Y), \neg p(Z, Y) \,\}, \{\, (X, Z) \,\}, \{\, X \mapsto \Gamma, Y \mapsto \Delta, Z \mapsto \Gamma \,\})$$

for some predicate p, variables X, Y, Z, and domains Γ and Δ. Then $\mathrm{Vars}(\{\, (Y, Z) \,\}) = \{\, Y, Z \,\}$, and $\mathrm{Vars}(c_1) = \mathrm{Vars}(c_2) = \{\, X, Y, Z \,\}$. Based on the domain maps of c_1 and c_2, we can infer that $p \in \Gamma \times \Delta$. All variables (in both clauses) that occur as the first argument to p are in Γ, and,

[3]Van den Broeck et al. (2011) refer to clauses as c-clauses.

likewise, all variables that occur as the second argument to p are in Δ. Therefore, ϕ 'type checks' as a valid formula.

There are two major differences between Definitions 1–3 and the corresponding concepts introduced by Van den Broeck et al. (2011). First, we decouple variable-to-domain assignments from constraints and move them to a separate function δ_c in Definition 2. Formalising these assignments as a function unveils the (previously implicit) assumption that each variable must be assigned to a domain. Second, while Van den Broeck et al. (2011) allow for equality constraints and constraints of the form $X \notin \Delta$ for some variable X and domain Δ, we exclude such constraints simply because they are not needed.

One can read a formula from Definition 3 as a sentence in a FOL. All variables in a clause are implicitly universally quantified (but note that variables are never shared among clauses), and all clauses in a formula are implicitly linked by a conjunction. Thus, formula ϕ from Example 1 reads as

$$(\forall X \in \Gamma.\, \forall Y, Z \in \Delta.\\ \quad Y \neq Z \implies \neg p(X, Y) \vee \neg p(X, Z)) \wedge\\ (\forall X, Z \in \Gamma.\, \forall Y \in \Delta.\\ \quad X \neq Z \implies \neg p(X, Y) \vee \neg p(Z, Y)).$$

Once domains are mapped to finite sets and constants to specific (and different) elements in those sets, a formula can be viewed as a set of conditions that the predicates (interpreted as relations) have to satisfy. Hence, FOMC is the problem of counting the number of combinations of relations that satisfy these conditions.

Example 2. Let ϕ be as in Example 1 and let $|\Gamma| = |\Delta| = 2$. There are $2^{2\times 2} = 16$ possible relations between Γ and Δ. Let us count how many of them satisfy the conditions imposed on predicate p. The empty relation does. All four relations of cardinality one do too. Finally, there are two relations of cardinality two that satisfy the conditions as well. Thus, the FOMC of ϕ (when $|\Gamma| = |\Delta| = 2$) is 7. Incidentally, the FOMC of ϕ counts partial injections. We will continue to use the problem of counting partial injections (and the formula from Example 1 specifically) as the main running example throughout the paper.

Notation. We write $\langle\rangle$ and $\langle x \rangle$ to denote an empty list and a list with one element x, respectively, and $|l|$ for the length of list l. We write $\nrightarrow$ for partial functions. Let S be a set of constraints or literals, V a set of variables, and x a variable or a constant. We write $S[x/V]$ to denote S with all occurrences of all variables in V replaced with x.

3 From Circuits to Graphs

A *first-order deterministic decomposable negation normal form computational graph* (FCG) is a (weakly connected) directed graph with a single source, node labels, and ordered outgoing edges. Node labels consist of two parts: the *type* and the *parameters*. The type of a node determines its out-degree.

Along with many node types defined previously (Van den Broeck 2011; Van den Broeck et al. 2011), CRANE uses three more:

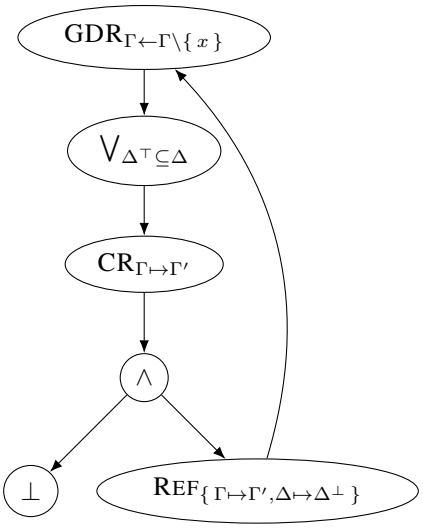

Figure 1: A simplified version of an FCG constructed by CRANE for the problem of counting partial injections from Example 1. Label $\bigvee_{\Delta^\top \subseteq \Delta}$ denotes set-disjunction, $\wedge$ denotes conjunction, and $\bot$ denotes a contradiction—see the work by Van den Broeck et al. (2011) for the descriptions of these node types. Here we omit some parameters as well as nodes whose only arithmetic effect is multiplication by one.

- a type for constraint removal denoted by CR,
- a type for generalised domain recursion denoted by GDR (both with out-degree one),
- and $\star$—a placeholder type (with out-degree zero) for nodes that are going to be replaced.

Similarly to Van den Broeck et al. (2011), we write T_p for an FCG that has a node with the label T_p (i.e., type T and parameter(s) p) and $\star$'s as all of its direct successors. See Fig. 1 for an example FCG. Its source node has out-degree 1, label $\mathrm{GDR}_{\Gamma \leftarrow \Gamma \setminus \{x\}}$, and type GDR.

Finally, we introduce a structure that represents a solution to a (W)FOMC problem while it is still being built. A *chip* is a pair (G, L), where G is an FCG, and L is a list of formulas, s.t. $|L|$ is equal to the number of $\star$'s in G. L contains formulas that still need to be compiled. Once a formula is compiled, it replaces one of the $\star$'s in G according to a set order. We say that an FCG is *complete* (i.e., it represents a *complete solution*) if it has no $\star$'s. Similarly, a chip is complete if its FCG is complete (or, equivalently, the list of formulas is empty).

4 New Compilation Rules

A *(compilation) rule* takes a formula and returns a set of chips. The cardinality of this set is the number of different ways in which the rule can be applied to the input formula. While FORCLIFT (Van den Broeck et al. 2011) heuristically chooses one of them, in an attempt to not miss a solution, CRANE returns them all. In particular, if a rule returns an empty set, then that rule does not apply to the formula.

Algorithm 1: The compilation rule for GDR nodes.

Input: formula ϕ, set of all relevant domains $\mathcal{D}$
Output: set of chips S

1 $S \leftarrow \emptyset$;
2 **foreach** *domain* $\Omega \in \mathcal{D}$ *s.t. there is* $c \in \phi$ *and* $X \in \mathrm{Vars}(L_c)$ *s.t.* $\delta_c(X) = \Omega$ **do**
3 $\phi' \leftarrow \emptyset$;
4 $x \leftarrow$ a new constant in domain Ω;
5 **foreach** *clause* $c = (L, C, \delta) \in \phi$ **do**
6 $V \leftarrow \{ X \in \mathrm{Vars}(L) \mid \delta(X) = \Omega \}$;
7 **foreach** *subset* $W \subseteq V$ *s.t.* $W^2 \cap C = \emptyset$ **and** $W \cap \{ X \in \mathrm{Vars}(C) \mid (X, y) \in C$ *for some constant* $y \} = \emptyset$ **do**
 `/* δ′ restricts δ to the new set of variables */`
8 $\phi' \leftarrow \phi' \cup \{ (L[x/W], C[x/W] \cup \{ (X, x) \mid (X \in V \setminus W) \}, \delta') \}$;
9 $S \leftarrow S \cup \{ (\mathrm{GDR}_{\Omega \leftarrow \Omega \setminus \{ x \}}, \langle \phi' \rangle) \}$;

4.1 Generalised Domain Recursion

The main idea behind domain recursion (both the original version by Van den Broeck (2011) and the one presented here) is as follows. Let $\Omega \in \mathcal{D}$ be a domain. Assuming that $\Omega \neq \emptyset$, pick some $x \in \Omega$. Then, for every variable $X \in \Omega$ that occurs in a literal, consider two possibilities: $X = x$ and $X \neq x$.

Example 3. Let ϕ be a formula with a single clause

$$(\{ \neg p(X, Y), \neg p(X, Z) \}, \{ (Y, Z) \}, \{ X \mapsto \Gamma, Y \mapsto \Delta, Z \mapsto \Delta \}).$$

Then we can introduce constant $x \in \Gamma$ and rewrite ϕ as $\phi' = \{ c_1, c_2 \}$, where

$$c_1 = (\{ \neg p(x, Y), \neg p(x, Z) \}, \{ (Y, Z) \}, \{ Y \mapsto \Delta, Z \mapsto \Delta \}),$$
$$c_2 = (\{ \neg p(X, Y), \neg p(X, Z) \}, \{ (X, x), (Y, Z) \}, \{ X \mapsto \Gamma', Y \mapsto \Delta, Z \mapsto \Delta \}),$$

and $\Gamma' = \Gamma \setminus \{ x \}$.

Van den Broeck (2011) imposes stringent preconditions on the input formula to ensure that the expanded version of the formula (as in Example 3) can be handled efficiently. For example, at least one clause must contain a variable that occurs in all of its atoms. The clauses in this expanded formula are then partitioned into three parts based on whether the transformation introduced constants or constraints or both. The preconditions ensure that these parts can be treated independently.

In contrast, GDR has only one precondition: for GDR to be applicable on domain $\Omega \in \mathcal{D}$, there must be at least one variable with domain Ω that is featured in a literal (and not just in constraints). Without such variables, GDR would have no effect on the formula. GDR is also simpler in that the expanded formula is left as-is to be handled by other compilation rules. Typically, after a few more rules are applied, a combination of CR and REF nodes introduces a cycle-inducing edge back to the GDR node, thus completing the definition of a recursive function. The GDR compilation rule is summarised as Algorithm 1 and explained in more detail using the example below.

Example 4. Let $\phi \coloneqq \{ c_1, c_2 \}$ be the formula from Example 1. While GDR is possible on both domains, here we illustrate how it works on Γ. Having chosen a domain, the algorithm iterates over the clauses of ϕ. Suppose line 5 picks $c = c_1$ as the first clause. Then, set V is constructed to contain all variables with domain $\Omega = \Gamma$ that occur in the literals of clause c. In this case, $V = \{ X \}$.

Line 7 iterates over all subsets $W \subseteq V$ of variables that can be replaced by a constant without resulting in evidently unsatisfiable formulas. We impose two restrictions on W. First, $W^2 \cap C = \emptyset$ ensures that there are no pairs of variables in W that are constrained to be distinct, since that would result in an $x \neq x$ constraint after substitution. Similarly, we want to avoid variables in W that have inequality constraints with constants: after the substitution, such constraints would transform into inequality constraints between two constants. In this case, both subsets of V satisfy these conditions, and line 8 generates two clauses for the output formula:

$$(\{ \neg p(X, Y), \neg p(X, Z) \}, \{ (Y, Z), (X, x) \}, \{ X \mapsto \Gamma, Y \mapsto \Delta, Z \mapsto \Delta \}),$$

from $W = \emptyset$ and

$$(\{ \neg p(x, Y), \neg p(x, Z) \}, \{ (Y, Z) \}, \{ Y \mapsto \Delta, Z \mapsto \Delta \})$$

from $W = V$.

When line 5 picks $c = c_2$, then $V = \{ X, Z \}$. The subset $W = V$ fails to satisfy the conditions on line 7 because of the $X \neq Z$ constraint. The other three subsets of V all generate clauses for ϕ'. Indeed, $W = \emptyset$ generates

$$(\{ \neg p(X, Y), \neg p(Z, Y) \}, \{ (X, Z), (X, x), (Z, x) \}, \{ X \mapsto \Gamma, Y \mapsto \Delta, Z \mapsto \Gamma \}),$$

$W = \{ X \}$ generates

$$(\{ \neg p(x, Y), \neg p(Z, Y) \}, \{ (Z, x) \}, \{ Y \mapsto \Delta, Z \mapsto \Gamma \}),$$

and $W = \{ Z \}$ generates

$$(\{ \neg p(X, Y), \neg p(x, Y) \}, \{ (X, x) \}, \{ X \mapsto \Gamma, Y \mapsto \Delta \}).$$

4.2 Constraint Removal

Recall that GDR on a domain Ω creates constraints of the form $X_i \neq x$ for some constant $x \in \Omega$ and family of variables $X_i \in \Omega$. Once certain conditions are satisfied, Algorithm 2 can eliminate these constraints and replace Ω with a new domain Ω', which can be interpreted as $\Omega \setminus \{ x \}$. These conditions (on line 2 of the algorithm) are that a constraint of the form $X \neq x$ exists for all variables $X \in \Omega$ across all clauses, and such constraints are the only place where x occurs. The algorithm then proceeds to construct the new formula by removing constraints (on line 6) and constructing a new domain map δ' that replaces Ω with Ω' (on line 7).

Algorithm 2: The compilation rule for CR nodes.

Input: formula ϕ, set of all relevant domains $\mathcal{D}$
Output: set of chips S

1 $S \leftarrow \emptyset$;
2 **foreach** *domain* $\Omega \in \mathcal{D}$ *and element* $x \in \Omega$ *s.t.* x *does not occur in any literal of any clause of* ϕ **and** *for each clause* $c = (L, C, \delta_c) \in \phi$ *and variable* $X \in \mathrm{Vars}(c)$, *either* $\delta_c(X) \neq \Omega$ *or* $(X, x) \in C$ **do**
3 add a new domain Ω' to $\mathcal{D}$;
4 $\phi' \leftarrow \emptyset$;
5 **foreach** *clause* $(L, C, \delta) \in \phi$ **do**
6 $C' \leftarrow \{ (a, b) \in C \mid b \neq x \}$;
7 $\delta' \leftarrow X \mapsto \begin{cases} \Omega' & \text{if } \delta(X) = \Omega \\ \delta(X) & \text{otherwise;} \end{cases}$
8 $\phi' \leftarrow \phi' \cup \{ (L, C', \delta') \}$
9 $S \leftarrow S \cup \{ (\mathrm{CR}_{\Omega \mapsto \Omega'}, \langle \phi' \rangle) \}$;

Example 5. Let $\phi = \{ c_1, c_2 \}$ be a formula with clauses

$$c_1 = (\{ \neg p(X, Y), \neg p(X, Z) \}, \{ (X, x), (Y, Z) \}, \{ X \mapsto \Gamma, Y \mapsto \Delta, Z \mapsto \Delta \}),$$
$$c_2 = (\{ \neg p(X, Y), \neg p(Z, Y) \}, \{ (X, x), (Z, X), (Z, x) \}, \{ X \mapsto \Gamma, Y \mapsto \Delta, Z \mapsto \Gamma \}).$$

Domain Γ and its element $x \in \Gamma$ satisfy the preconditions for CR. The rule introduces a new domain Γ' and transforms ϕ to $\phi' = (c'_1, c'_2)$, where

$$c'_1 = (\{ \neg p(X, Y), \neg p(X, Z) \}, \{ (Y, Z) \}, \{ X \mapsto \Gamma', Y \mapsto \Delta, Z \mapsto \Delta \}),$$
$$c'_2 = (\{ \neg p(X, Y), \neg p(Z, Y) \}, \{ (Z, X) \}, \{ X \mapsto \Gamma', Y \mapsto \Delta, Z \mapsto \Gamma' \}).$$

4.3 Identifying Opportunities for Recursion

Notation. Let Doms be a function that maps any clause or formula to the set of domains used within, and let $\mathrm{dom}(f)$ denote the domain of function f.

Hashing. We use (integer-valued) hash functions to discard pairs of formulas that are too different for recursion to work. The hash code of a clause $c = (L, C, \delta_c)$ (denoted by $\#c$) combines the hash codes of the sets of constants and predicates in c, the numbers of positive and negative literals, the number of inequality constraints $|C|$, and the number of variables $|\mathrm{Vars}(c)|$. The hash code of a formula ϕ combines the hash codes of all its clauses and is denoted by $\#\phi$.

Caching. FORCLIFT (Van den Broeck et al. 2011) uses a cache to check if a formula is identical to one of the formulas that have already been fully compiled. To facilitate recursion, we extend the caching scheme to include formulas that have been encountered but not fully compiled yet. Formally, we define a *cache* to be a map from integers (e.g., hash codes) to sets of pairs of the form (ϕ, v), where ϕ is a formula, and v is an FCG node.

Algorithm 3: The compilation rule for REF nodes.

Input: formula ϕ, cache C
Output: a set of chips

1 **foreach** *formula and node* $(\psi, v) \in C(\#\phi)$ **do**
2 $\rho \leftarrow$ `r` (ϕ, ψ) ;
3 **if** $\rho \neq$ `null` **then return** $\{ (\mathrm{REF}_\rho(v), \langle\rangle) \}$;
4 **return** $\emptyset$;
5 **Function** `r` (*formula* ϕ, *formula* ψ, *map* $\rho = \emptyset$) **:**
6 **if** $|\phi| \neq |\psi|$ **or** $\#\phi \neq \#\psi$ **then return** `null`;
7 **if** $\phi = \emptyset$ **then return** ρ;
8 **foreach** *clause* $c \in \psi$ **do**
9 **foreach** *clause* $d \in \phi$ *s.t.* $\#d = \#c$ **do**
10 **forall** $\gamma \in$ `genMaps` (c, d, ρ) **do**
11 $\rho' \leftarrow$ `r` $(\phi \setminus \{ d \}, \psi \setminus \{ c \}, \rho \cup \gamma)$;
12 **if** $\rho' \neq$ `null` **then return** ρ';
13 **return** `null`;

Algorithm 3 describes the compilation rule for creating REF nodes. For every formula ψ in the cache s.t. $\#\psi = \#\phi$, function `r` is called to check whether a recursive call is feasible. If it is, `r` returns a (total) map $\rho\colon \mathrm{Doms}(\psi) \to \mathrm{Doms}(\phi)$ that shows how ψ can be transformed into ϕ by replacing each domain $\Omega \in \mathrm{Doms}(\psi)$ with $\rho(\Omega) \in \mathrm{Doms}(\phi)$. Otherwise, `r` returns `null` to signify that ϕ and ψ are too different for recursion to work. This happens if ϕ and ψ (or their subformulas explored in recursive calls) are structurally different (i.e., the numbers of clauses or the hash codes fail to match) or if a clause of ψ cannot be paired with a sufficiently similar clause of ϕ. Function `r` works by iterating over pairs of clauses of ϕ and ψ that have the same hash codes. For every pair of sufficiently similar clauses, `r` calls itself on the remaining clauses until the map $\rho\colon \mathrm{Doms}(\psi) \nrightarrow \mathrm{Doms}(\phi)$ becomes total, and all clauses are successfully coupled.

Function `genMaps` checks the compatibility of a pair of clauses. It considers every possible bijection $\beta\colon \mathrm{Vars}(c) \to \mathrm{Vars}(d)$ and map $\gamma\colon \mathrm{Doms}(c) \to \mathrm{Doms}(d)$ s.t.

$$\begin{array}{ccc} \mathrm{Vars}(c) & \xrightarrow{\beta} & \mathrm{Vars}(d) \\ {\scriptstyle \delta_c}\downarrow & & \downarrow{\scriptstyle \delta_d} \\ \mathrm{Doms}(c) & \xrightarrow{\gamma} & \mathrm{Doms}(d) \\ \downarrow & & \downarrow \\ \mathrm{Doms}(\psi) & \xrightarrow[\rho]{} & \mathrm{Doms}(\phi). \end{array}$$

commutes, and c becomes equal to d when its variables are replaced according to β and its domains replaced according to γ. The function then returns each such γ as soon as possible. The commutativity check also ensures that $\rho \cup \gamma$ is possible on line 11 of Algorithm 3. The resulting (partial) function $\rho \cup \gamma$ is the unique function s.t. $\rho \cup \gamma|_{\mathrm{dom}(\rho)} = \rho$, and $\rho \cup \gamma|_{\mathrm{dom}(\gamma)} = \gamma$. The operation is defined when $\rho|_{\mathrm{dom}(\rho) \cap \mathrm{dom}(\gamma)} = \gamma|_{\mathrm{dom}(\rho) \cap \mathrm{dom}(\gamma)}$.

5 How to Interpret an FCG

When FORCLIFT (Van den Broeck et al. 2011) compiles a WFOMC instance into a circuit, each node type encodes an arithmetic operation on its inputs and parameters. These operations are then immediately performed while traversing the circuit. With CRANE, the interpretation of an FCG is a collection of functions. Each function has (some) domain sizes as parameters and may contain recursive calls to other functions, including itself. While there may be any number of subsidiary functions, there is always one main function that can be called with the sizes of the domains of the input formula as arguments. Henceforth, this function is always called f, and it is defined by the source node.

The interpretation of a node is decided by its type. Here we describe the interpretations of new (or significantly changed) types and refer the reader to previous work (Van den Broeck et al. 2011) for information on other types. The interpretation of a CR or a GDR node is simply the interpretation of its only direct successor. Obviously, $\star$ nodes have no interpretation as incomplete FCGs are not meant to be interpreted. The interpretation of a REF node is a function call. The direct successor of the REF node (say, v) then must introduce a function. The parameters of this function are the sizes of all domains used by nodes reachable from v.

Example 6. Consider the FCG from Fig. 1. The input formula (i.e., the formula from Example 1) has two domains: Γ and Δ. Thus, the interpretation of the FCG is a function $f\colon \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{R}_{\geq 0}$. Let $m \coloneqq |\Gamma|$, and $n \coloneqq |\Delta|$. The node labelled $\bigvee_{\Delta^\top \subseteq \Delta}$ tells us that $f(m,n) = \sum_{l=0}^{n} \binom{n}{l} \square$, where $\square$ is the interpretation of the remaining subgraph, and l iterates over all possible sizes of $\Delta^\top$. It also creates two subdomains $\Delta^\top, \Delta^\bot \subseteq \Delta$ that partition Δ, i.e., as the size of $\Delta^\top$ increases, the size of $\Delta^\bot$ correspondingly decreases. Nodes labelled $\wedge$ correspond to multiplication. Therefore, $f(m,n) = \sum_{l=0}^{n} \binom{n}{l} \diamondsuit \times \heartsuit$, where $\diamondsuit$ is the interpretation of the contradiction (i.e., $\bot$) node, and $\heartsuit$ is the interpretation of the REF node.

A contradiction node with clause c as a parameter is interpreted as one if the clause has groundings and zero otherwise. In this case, the parameter is $c = (\emptyset, \{(X,Y)\}, \{X \mapsto \Delta^\top, Y \mapsto \Delta^\top\})$ (not shown in Fig. 1), which can be read as $\forall X, Y \in \Delta^\top.\, X \neq Y \implies \bot$, i.e., $\forall X, Y \in \Delta^\top.\, X = Y$. This latter sentence is true if and only if $|\Delta^\top| < 2$. Therefore, we can use the Iverson bracket notation to write

$$\diamondsuit = [l < 2] \coloneqq \begin{cases} 1 & \text{if } l < 2 \\ 0 & \text{otherwise.} \end{cases}$$

It remains to interpret the REF node. Parameter $\{\Gamma \mapsto \Gamma', \Delta \mapsto \Delta^\bot\}$ tells us that the interpretation of the REF node should be the same as that of the source node, but with domains Γ and Δ replaced with Γ' and $\Delta^\bot$, respectively. Domain Γ' was created by the CR rule applied on Γ, so $|\Gamma'| = m - 1$. Now $\Delta^\bot = \Delta \setminus \Delta^\top$, and $|\Delta^\top| = l$, so $|\Delta^\bot| = n - l$. Thus, the interpretation of the REF node is a recursive call to $f(m-1, n-l)$. Therefore,

$$\begin{aligned} f(m,n) &= \sum_{l=0}^{n} \binom{n}{l} [l < 2] f(m-1, n-l) \\ &= f(m-1,n) + n f(m-1, n-1). \end{aligned} \tag{1}$$

To use this recursive function to compute the model count of the input formula for any domain sizes, one just needs to find the base cases $f(0,n)$ and $f(m,0)$ for all $m, n \in \mathbb{N}_0$.

6 Empirical Results

We compare CRANE and FORCLIFT (Van den Broeck et al. 2011) on their ability to count various kinds of functions. This class of instances is chosen because of its simplicity and ubiquity and the inability of state-of-the-art WFOMC algorithms to solve many such instances. Note that other WFOMC algorithms—L2C (Kazemi and Poole 2016) and probabilistic theorem proving (Gogate and Domingos 2016)—are unable to solve any of the instances that FORCLIFT fails on. We begin by describing how such function-counting problems can be expressed in FOL. FORCLIFT then translates these sentences in FOL to formulas as defined in Definition 3.

Let $p \in \Gamma \times \Delta$ be a predicate. To restrict all relations representable by p to just functions from Γ to Δ, in FOL one might write

$$\forall X \in \Gamma.\, \forall Y, Z \in \Delta.\, p(X,Y) \wedge p(X,Z) \implies Y = Z$$

and

$$\forall X \in \Gamma.\, \exists Y \in \Delta.\, p(X,Y). \tag{2}$$

The former sentence says that one element of Γ can map to at *most* one element of Δ, and the latter sentence says that each element of Γ must map to at *least* one element of Δ. One can then add

$$\forall X, Z \in \Gamma.\, \forall Y \in \Delta.\, p(X,Y) \wedge p(Z,Y) \implies X = Z$$

to restrict p to injections or

$$\forall Y \in \Delta.\, \exists X \in \Gamma.\, p(X,Y)$$

to ensure surjectivity or remove Eq. (2) to consider partial functions. Lastly, one can replace all occurrences of Δ with Γ to model endofunctions (i.e., functions with the same domain and codomain) instead.

In our experiments, we consider all sixteen combinations of these properties, i.e., injectivity, surjectivity, partiality, and endo-. FORCLIFT is always run until it terminates. CRANE is run until either five solutions are found or the search tree reaches height 6 (which happens in at most a few seconds). If successful, FORCLIFT generates a circuit, and CRANE generates one or more (complete) FCGs. In both cases, we manually convert the resulting graphs into definitions of functions as described in Section 5. We then assess the complexity of each solution and pick the best if CRANE returns several solutions of varying complexities. When assessing the complexity of each such definition, we make two assumptions. First, we can compute the binomial coefficient $\binom{n}{k}$ in $\Theta(nk)$ time. Second, techniques such as dynamic programming and memoization are used to avoid recomputing the same binomial coefficient or function call multiple times.

Function Class			Asymptotic Complexity of Counting		
Partial	Endo-	Class	Best Known	With FORCLIFT	With CRANE
✓/✗	✓/✗	Functions	$\log m$	m	m
✗	✗	Surjections	$n \log m$	$m^3 + n^3$	$m^3 + n^3$
✗	✓		$m \log m$	m^3	m^3
✓	✗		Same as injections from Δ to Γ		
✓	✓		Same as endo-injections		
✗	✗	Injections	m	—	mn
✗	✓		m	—	m^3
✓	✗		$\min\{m, n\}^2$	—	mn
✓	✓		m^2	—	—
✗	✗	Bijections	m	—	m
✗	✓		Same as (partial) (endo-)injections		
✓	✓/✗				

Table 1: The worst-case complexity of counting various types of functions. Here, m is the size of domain Γ, and n is the size of domain Δ. All asymptotic complexities are in $\Theta(\cdot)$. A dash means that no complete solution was found.

The experimental results are in Table 1. Previous work often compares WFOMC algorithms by running them on a few instances with increasing domain sizes and measuring runtime (Van den Broeck 2011; Van den Broeck et al. 2011; Van den Broeck and Davis 2012). However, we can do better than that and identify the exact worst-case asymptotic complexity of a solution produced by either CRANE or FORCLIFT. The best-known asymptotic complexity for computing total surjections is by Earnest (2018). All other best-known complexity results are inferred from the formulas and programs on the on-line encyclopedia of integer sequences (OEIS Foundation Inc. 2022). On instances that could already be solved by FORCLIFT, the two algorithms perform equally well. However, CRANE can also solve all but one instances that FORCLIFT fails on in at most cubic time.

Let us examine the case of counting partial (non-endomorphic) injections more closely. The FCG in Fig. 1 and Example 6 counts partial injections and is responsible for the mn entry in the table. For a complete solution, Eq. (1) must be combined with the base case $f(0, n) = 1$ for all $n \in \mathbb{N}_0$. This base case says that the empty partial map is the only partial injection with an empty domain. Finally, note that $f(m, n)$ can be evaluated in $\Theta(mn)$ time by a dynamic programming algorithm that computes $f(i, j)$ for all $i = 0, \ldots, m$ and $j = 0, \ldots, n$.

7 Conclusion and Future Work

In this paper, we showed how a state-of-the-art (W)FOMC algorithm can be empowered by generalising domain recursion and adding support for cycles in the graph that encodes a solution. To construct such graphs, CRANE supplements FORCLIFT (Van den Broeck et al. 2011) with three new compilation rules. Our experiments revealed a range of counting problems that are liftable to CRANE but not to any other (W)FOMC algorithm. Although in this paper we focus on unweighted counting, FORCLIFT's support for weights trivially transfers to CRANE as well. The common thread across these newly liftable problems is (partial) injectivity. Thus, we can formulate the following conjecture.

Conjecture 1. *Let* IFO^2 *be the class of formulas in FOL that contain clauses with at most two variables as well as any number of copies of*

$$(\forall X \in \Gamma.\ \forall Y, Z \in \Delta.\ Y \neq Z \implies \neg p(X, Y) \vee \neg p(X, Z)) \wedge$$
$$(\forall X, Z \in \Gamma.\ \forall Y \in \Delta.\ X \neq Z \implies \neg p(X, Y) \vee \neg p(Z, Y))$$

for some predicate p *and domains* Γ *and* Δ*. Then* IFO^2 *is liftable by* CRANE*.*

Recall that C^2 is the class of formulas with counting quantifiers and at most two variables. C^2 was recently shown to be liftable (Kuzelka 2021) but without providing a usable (W)FOMC algorithm. Since the tasks of counting injections and bijections fall into the C^2 fragment, we can conjecture the following.

Conjecture 2. C^2 *is liftable by* CRANE *by either reformulating formulas in* C^2 *to avoid counting quantifiers or extending* CRANE *to support them.*

However, the most important direction for future work is to fully automate this new process. First, we need an algorithm that transforms FCGs into definitions of functions. Formalising this process would also allow us to prove the correctness of the new compilation rules in constructing FCGs that indeed compute the right (weighted) model counts. Second, these definitions must be simplified before they can be used, perhaps by a computer algebra system. Third, we need a way to find the base cases for the recursive definitions provided by CRANE. Fourth, since the first solution found by CRANE is not always optimal in terms of its complexity, an automated way to determine the asymptotic complexity of a solution would be helpful as well. Achieving these goals would enable CRANE to automatically constructing efficient ways to compute various functions and sequences. In addition to the potential impact on areas of artificial intelligence (AI) such as statistical relational AI (De Raedt et al. 2016), CRANE could be beneficial to research in combinatorics as well (Barvínek et al. 2021).

References

Barvínek, J.; van Bremen, T.; Wang, Y.; Zelezný, F.; and Kuzelka, O. 2021. Automatic Conjecturing of P-Recursions Using Lifted Inference. In Katzouris, N.; and Artikis, A., eds., *Inductive Logic Programming - 30th International Conference, ILP 2021, Virtual Event, October 25-27, 2021, Proceedings*, 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 Milo, T.; and Calvanese, D., eds., *Proceedings of the 34th ACM Symposium on Principles of Database Systems, PODS 2015, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, 313–328. ACM.

Chavira, M.; and Darwiche, A. 2008. On probabilistic inference by weighted model counting. *Artif. Intell.*, 172(6-7): 772–799.

De Raedt, L.; Kersting, K.; Natarajan, S.; and Poole, D. 2016. *Statistical Relational Artificial Intelligence: Logic, Probability, and Computation*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

De Raedt, L.; and Kimmig, A. 2015. Probabilistic (logic) programming concepts. *Machine Learning*, 100(1): 5–47.

Earnest, M. 2018. An efficient way to numerically compute Stirling numbers of the second kind? https://scicomp.stackexchange.com/q/30049. Accessed: 2022-07-23.

Gatterbauer, W.; and Suciu, D. 2015. Approximate Lifted Inference with Probabilistic Databases. *Proc. VLDB Endow.*, 8(5): 629–640.

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.

Jaeger, M.; and Van den Broeck, G. 2012. Liftability of probabilistic inference: Upper and lower bounds. In *Proceedings of the 2nd international workshop on statistical relational AI*.

Kazemi, S. M.; Kimmig, A.; Van den Broeck, G.; and Poole, D. 2016. New Liftable Classes for First-Order Probabilistic Inference. In Lee, D. D.; Sugiyama, M.; von Luxburg, U.; Guyon, I.; and Garnett, R., eds., *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, 3117–3125.

Kazemi, S. M.; and Poole, D. 2016. Knowledge Compilation for Lifted Probabilistic Inference: Compiling to a Low-Level Language. In Baral, C.; Delgrande, J. P.; and Wolter, F., eds., *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifteenth International Conference, KR 2016, Cape Town, South Africa, April 25-29, 2016*, 561–564. AAAI Press.

Kersting, K. 2012. Lifted Probabilistic Inference. In De Raedt, L.; Bessiere, C.; Dubois, D.; Doherty, P.; Frasconi, P.; Heintz, F.; and Lucas, P. J. F., eds., *ECAI 2012 - 20th European Conference on Artificial Intelligence. Including Prestigious Applications of Artificial Intelligence (PAIS-2012) System Demonstrations Track, Montpellier, France, August 27-31 , 2012*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, 33–38. IOS Press.

Kimmig, A.; Mihalkova, L.; and Getoor, L. 2015. Lifted graphical models: a survey. *Mach. Learn.*, 99(1): 1–45.

Kuusisto, A.; and Lutz, C. 2018. Weighted model counting beyond two-variable logic. In Dawar, A.; and Grädel, E., eds., *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, 619–628. ACM.

Kuzelka, 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 *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*, 5817–5824. AAAI Press.

OEIS Foundation Inc. 2022. The On-Line Encyclopedia of Integer Sequences. Published electronically at https://oeis.org.

Richardson, M.; and Domingos, P. M. 2006. Markov logic networks. *Mach. Learn.*, 62(1-2): 107–136.

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.

van Bremen, T.; and Kuzelka, O. 2020. Approximate Weighted First-Order Model Counting: Exploiting Fast Approximate Model Counters and Symmetry. In Bessiere, C., ed., *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, 4252–4258. ijcai.org.

van Bremen, T.; and Kuzelka, O. 2021a. Faster lifting for two-variable logic using cell graphs. In de Campos, C. P.; Maathuis, M. H.; and Quaeghebeur, E., eds., *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence, UAI 2021, Virtual Event, 27-30 July 2021*, volume 161 of *Proceedings of Machine Learning Research*, 1393–1402. AUAI Press.

van Bremen, T.; and Kuzelka, O. 2021b. Lifted Inference with Tree Axioms. In Bienvenu, M.; Lakemeyer, G.; and Erdem, E., eds., *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning, KR 2021, Online event, November 3-12, 2021*, 599–608.

Van den Broeck, G. 2011. On the Completeness of First-Order Knowledge Compilation for Lifted Probabilistic Inference. In Shawe-Taylor, J.; Zemel, R. S.; Bartlett, P. L.; Pereira, F. C. N.; and Weinberger, K. Q., eds., *Advances*

in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain, 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 de Freitas, N.; and Murphy, K. P., eds., *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence, Catalina Island, CA, USA, August 14-18, 2012*, 131–141. AUAI Press.

Van den Broeck, G.; and Davis, J. 2012. Conditioning in First-Order Knowledge Compilation and Lifted Probabilistic Inference. In Hoffmann, J.; and Selman, B., eds., *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada*. 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 Walsh, T., ed., *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, 2178–2185. IJCAI/AAAI.