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 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 are severely 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 us to find efficient solutions to counting fundamental structures such as injections and bijections.

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. 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).

Traditionally in computational complexity theory, a problem is *tractable* if it can be solved in time polynomial in the size of the instance. 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, more and more 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. 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 which 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.

[1] The problem of computing the factorial can be described using two variables and counting quantifiers, so it is known to be liftable in theory (Kuzelka 2021).

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 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 written in Scala. FORCLIFT works in two stages: compilation and evaluation/propagation.[2] In the first part, various *(compilation) rules* are applied to the input (or some derivative) formula, gradually constructing a circuit. In the second part, the weights of the instance (and sometimes constants) are propagated through the circuit, computing the WMC.

TODO: improve the wording of the paragraph below.

The main conceptual difference between CRANE and FORCLIFT is that we utilise labelled directed graphs instead of circuits. The cycles in these graphs represent recursive calls. Suppose the original formula ϕ depends on a domain of size $n \in \mathbb{N}$. Our generalised version of domain recursion transforms ϕ into a different formula ψ that depends on a domain of size $n - 1$. After some number of subsequent transformations, the algorithm identifies that a solution to ψ can be constructed in part by finding a solution to a version of ϕ where the domain of size n is replaced by a domain of size $n - 1$. Recognising ϕ from before, we can add a cycle-forming edge to the graph, which can be interpreted as function f relying on $f(n-1)$ to compute $f(n)$.

To construct such graphs, we introduce a novel search algorithm that replaces greedy search used by FORCLIFT and three compilation rules that work together to construct cyclic graphs. We begin with Section 2 where we define the representation used for sentences in FOL as well as discuss caching and some notational conventions. Then, at the beginning of Section 3 we formally define the graphs that replace circuits in representing a solution to a (W)FOMC problem and discuss some related notions. Section 4 introduces the new compilation rules:

- a generalisation of domain recursion by Van den Broeck (2011),
- *constraint removal*—a formula transformation rule that removes the constraints introduced by generalised domain recursion (GDR), introducing a new (smaller) domain instead,
- and a rule for creating recursive calls.

In a typical solution produced by CRANE, these rules are executed in exactly this order (interspersed with some other rules), with the edge that represents a recursive call pointing back to the node introduced by GDR. In Section 5, we informally discuss how a graph can be interpreted as a collection of (potentially recursive) functions. Finally, 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 instances that FORCLIFT fails on as well.

[2]There is also an intermediate stage called *smoothing* that takes place between compilation and evaluation. As our changes to smoothing are quite elementary, we do not discuss them to not distract the reader from the main contributions of this paper.

2 Preliminaries

In this section, we describe our format for FOMC instances, introduce some notation, and discuss our caching scheme, which is used to identify possibilities for a recursive call. Note that although the focus of this paper is on unweighted model counting, FORCLIFT's (Van den Broeck et al. 2011) support for weights trivially transfers to CRANE as well.

Our representation of FOMC instances is heavily based on the format used internally by FORCLIFT, some aspects of which are described by Van den Broeck et al. (2011). FORCLIFT is able to translate sentences in a variant of function-free many-sorted FOL with equality to this internal format. An *atom* is $p(t_1, \ldots, t_n)$ for some predicate p and terms $t_1, \ldots, t_n$. Here, $n \in \mathbb{N}_0$ is the *arity* of p. 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 domains; note that this set expands during compilation.

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 x, predicate p, and domains a and b, we write, e.g., $x \in a$ and $p \in a \times b$ to denote that x is associated with a, and p is associated with a and b (in that order).

Definition 3 (Formula)**.** A *formula* (called a c-theory by Van den Broeck et al. (2011)) is a set of clauses such that all constraints and atoms 'type check' with respect to domains.

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

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

for some predicate p, variables X, Y, Z, and domains a and b. 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 $p \in a \times b$. All variables (in both clauses) that occur as the first argument to p are in a, and likewise all variables that occur as the second argument to p are in b. Therefore, ϕ 'type checks' as a valid formula.

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

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 $X \notin d$ for some variable X and domain d, we exclude such constraints simply because they are not needed.

One can read a formula as in 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 conjunction. Thus, formula ϕ from Example 1 reads as

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

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.[4] 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 $|a| = |b| = 2$. There are $2^{2\times 2} = 16$ possible relations between a and b. 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 $|a| = |b| = 2$) 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 $|l|$ to denote the length of list l. Let S be a set of constraints or literals, V a set of variables, and x either a variable or a constant. We write $S[x/V]$ to denote S with all occurrences of all variables in V replaced with x.[5]

3 From Circuits to Graphs

We begin this section by formally defining the graphs that CRANE uses as a generalisation of circuits. Then, Section 4 describes three new compilation rules.

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.[6] We denote an FCG as $G = (V, s, N^+, \tau)$, where V is the set of nodes, and $s \in V$ is the unique source. Function N^+ maps each node in V to a *list* of its direct successors. Node labels consist of two parts: the *type* and the *parameters*. To avoid clutter, we leave the parameters implicit and let τ denote the node-labelling function that maps each node in V to its type. For each node $v \in V$, the length of list $N^+(v)$ (i.e., the out-degree of v) is determined by its type $\tau(v)$. Most of the types are as in previous work (Van den Broeck 2011; Van den Broeck et al. 2011). The type for non-tree-like edges (denoted by REF), while used before, is extended to contain the information necessary to support recursive calls. We also add three new types:

- 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.

When drawing an FCG, we order outgoing edges from left to right, write node labels directly on the nodes, and omit irrelevant labels and/or parameters. See Fig. 1 for an example FCG. Its source node has out-degree 1 (i.e., $|N^+(s)| = 1$), label $\text{GDR}_{a \leftarrow a \setminus \{x\}}$, and type GDR (i.e., $\tau(s) = \text{GDR}$).

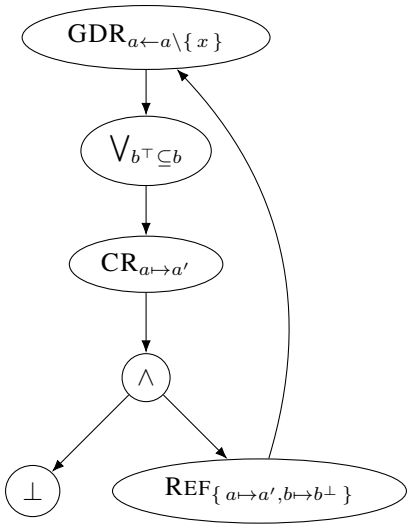
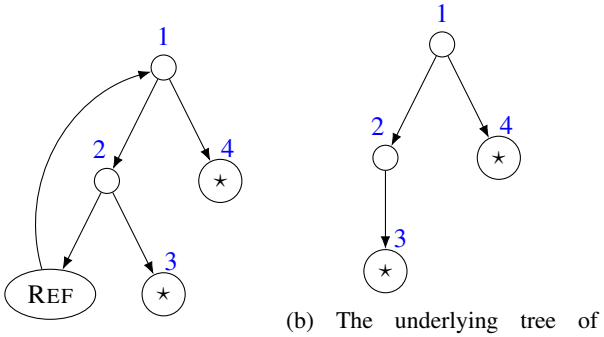


Figure 1: A simplified version of an FCG constructed by CRANE for the problem of counting partial injections from Example 1. Label $\bigvee_{b^\top \subseteq b}$ denotes set-disjunction, $\wedge$ denotes conjunction, and $\bot$ denotes a contradiction—see the work by Van den Broeck et al. (2011) for the descriptions of these node types. Here we omit nodes whose only arithmetic effect is multiplication by one. Some of these nodes play an important role in the weighted version of the problem whereas others are remnants of the interaction between compilation rules and the way in which FORCLIFT handles existential quantifiers.

[4] If some domain is not big enough to contain all of its constants, the formula is unsatisfiable.

[5] Note that if (X, Y) is a two-variable constraint, substituting a constant c for X would result in (c, Y), which would have to be rewritten as (Y, c) to fit the definition of a constraint.

[6] Note that imposing an ordering on outgoing edges is just a limited version of edge labelling.



(a) An example FCG (b) The underlying tree of Fig. 2a

Figure 2: An FCG and its underlying tree. The integers in blue denote the pre-order traversal of the underlying tree.

Similarly to Van den Broeck et al. (2011), we write T_p for an FCG that has a node with label T_p (i.e., type T and parameter(s) p) and $\star$'s as all of its direct successors. In particular, as an FCG, $\star$ denotes $(\{\, s \,\}, s, \{\, s \mapsto \langle\rangle \,\}, \{\, s \mapsto \star \,\})$, i.e., an FCG with just one node of type $\star$ and no edges. We write $T_p(v)$ for an FCG with one edge from a node labelled T_p to some other node v (and no other nodes or edges).

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, such that $|L|$ is equal to the number of $\star$'s in G. The intuition behind this definition is that L contains formulas that still need to be compiled. Once a formula is compiled, it replaced one of the $\star$'s in G. 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). Let the *underlying tree* of G be the induced subgraph of G that omits all REF nodes.[7] Then we can define an implicit bijection between the formulas in L and the $\star$'s in G according to the order in which elements of L are listed and the pre-order traversal of the underlying tree of G. For example, if G is as in Fig. 2a (with its underlying tree in Fig. 2b), then $|L| = 2$. Moreover, the first element of L is associated with the $\star$ labelled 3, and the second with the one labelled 4.

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 is not applicable to the formula, and the algorithm continues with the next rule.

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 $d \in \mathcal{D}$ be a domain. Assuming that $d \neq \emptyset$, pick some $x \in d$. Then, for every variable X associated with domain d that occurs in a literal, consider two possibilities: $X = x$ and $X \neq x$.

[7] Subsequently presented algorithms ensure that the underlying tree is guaranteed to be a tree.

Algorithm 1: The compilation rule for GDR nodes.

Input: formula ϕ
Output: set of chips S

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

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

$$(\{\, \neg p(X, Y), \neg p(X, Z) \,\}, \{\, (Y, Z) \,\}, \{\, X \mapsto a, Y \mapsto b, Z \mapsto b \,\}).$$

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

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

and $a' = a \setminus \{\, x \,\}$.

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. The clauses in this expanded formula are then partitioned into three parts based on whether the transformation introduced constants or constraints or both. The aforementioned conditions ensure that these parts can be treated independently.

In contrast, GDR has only one precondition: for GDR to be applicable on domain $d \in \mathcal{D}$, there must be at least one variable with domain d 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 loop 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 $\phi := \{ c_1, c_2 \}$ be the formula from Example 1. While GDR is possible on both domains, here we illustrate how it works on a. 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 $d = a$ 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 formulas that are evidently unsatisfiable. 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 a $x \neq x$ constraint after substitution. Similarly, we want to avoid variables in W that have inequality constraints with constants: after 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:

$$(\{ \neg p(X,Y), \neg p(X,Z) \}, \{ (Y,Z), (X,x) \}, \{ X \mapsto a, Y \mapsto b, Z \mapsto b \}),$$

from $W = \emptyset$ and

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

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

$$(\{ \neg p(X,Y), \neg p(Z,Y) \}, \{ (X,Z), (X,x), (Z,x) \}, \{ X \mapsto a, Y \mapsto b, Z \mapsto a \}),$$

$W = \{ X \}$ generates

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

and $W = \{ Z \}$ generates

$$(\{ \neg p(X,Y), \neg p(x,Y) \}, \{ (X,x) \}, \{ X \mapsto a, Y \mapsto b \}).$$

4.2 Constraint Removal

Recall that GDR on a domain d creates constraints of the form $X_i \neq x$ for some constant $x \in d$ and family of variables $X_i \in d$. Once certain conditions are satisfied, Algorithm 2 can eliminate these constraints and replace d with a new domain d', which can be interpreted as $d \setminus \{ x \}$. These conditions (on line 2 of the algorithm) are that a constraint of the form $X \neq e$ exists for all variables $X \in d$ across all clauses, and such constraints are the only place where e 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 d with d' (on line 7).

Example 5. Let $\phi = \{ c_1, c_2, c_3 \}$ be a formula with clauses

$$c_1 = (\emptyset, \{ (Y,X) \}, \{ X \mapsto b^\top, Y \mapsto b^\top \}),$$
$$c_2 = (\{ \neg p(X,Y), \neg p(X,Z) \}, \{ (X,x), (Y,Z) \}, \{ X \mapsto a, Y \mapsto b^\perp, Z \mapsto b^\perp \}),$$
$$c_3 = (\{ \neg p(X,Y), \neg p(Z,Y) \}, \{ (X,x), (Z,X), (Z,x) \}, \{ X \mapsto a, Y \mapsto b^\perp, Z \mapsto a \}).$$

Algorithm 2: The compilation rule for CR nodes.

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

1 $S \leftarrow \emptyset$;
2 **foreach** *domain* $d \in \mathcal{D}$ *and element* $x \in d$ *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* $v \in \mathrm{Vars}(c)$, *either* $\delta_c(v) \neq d$ *or* $(v, x) \in C$ **do**
3 add a new domain d' 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 $\delta' \leftarrow v \mapsto \begin{cases} d' & \text{if} \delta(v) = d \\ \delta(v) & \text{otherwise;} \end{cases}$
8 $\phi' \leftarrow \phi' \cup \{ (L, C', \delta') \}$
9 $S \leftarrow S \cup \{ (\mathrm{CR}_{d \mapsto d'}, \langle \phi' \rangle) \}$;

Domain a and with its element $x \in a$ satisfy the preconditions for constraint removal. The rule introduces a new domain a' and transforms ϕ to $\phi' = (c'_1, c'_2, c'_3)$, where

$$c'_1 = c_1,$$
$$c'_2 = (\{ \neg p(X,Y), \neg p(X,Z) \}, \{ (Y,Z) \}, \{ X \mapsto a', Y \mapsto b^\perp, Z \mapsto b^\perp \}),$$
$$c'_3 = (\{ \neg p(X,Y), \neg p(Z,Y) \}, \{ (Z,X) \}, \{ X \mapsto a', Y \mapsto b^\perp, Z \mapsto a' \}).$$

4.3 Identifying Opportunities for Recursion

TODO: write a short informal summary

5 How to Interpret an FCG

When FORCLIFT (Van den Broeck et al. 2011) compiles a WFOMC instance into a circuit, each gate type encodes an arithmetic operation on its inputs and parameters. These operations are then immediately performed while traversing the circuit and using domain sizes and weights as the initial inputs. 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. Both CR and GDR nodes do not contribute anything to the definitions of functions—the interpretation of such a node is simply the interpretation of its only direct successor. Obviously, $\star$ nodes also have no interpretation, although for a different reason: 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. Let us use the FCG from Fig. 1 as an example. The input formula (i.e., the formula in Example 1) has two domains: a and b. Thus, the interpretation of the FCG is a function $f\colon \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{R}_{\geq 0}$. Let $m \coloneqq |a|$, and $n \coloneqq |b|$. The node labelled $\bigvee_{b^\top \subseteq b}$ 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 $b^\top$. It also creates two subdomains $b^\top, b^\bot \subseteq b$ that partition b, i.e., as the size of $b^\top$ increases, the size of $b^\bot$ 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 $c = (\emptyset, \{(X,Y)\}, \{X \mapsto b^\top, Y \mapsto b^\top\})$ (not shown in Fig. 1), which can be read as $\forall X, Y \in b^\top.\, X \neq Y \implies \bot$, i.e., $\forall X, Y \in b^\top, X = Y$. This latter sentence is true if and only if $|b^\top| < 2$. 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 $\{a \mapsto a', b \mapsto b^\bot\}$ tells us that the interpretation of the REF node should be the same as that of the source node, but with domains a and b replaced with a' and $b^\bot$, respectively. Domain a' was created by a constraint removal rule applied on a, so $|a'| = m-1$. Now $b^\bot = b \backslash b^\top$, and $|b^\top| = l$, so $|b^\bot| = n-l$. Thus, the interpretation of the REF node is a recursive call to $f(m-1, n-l)$. Therefore,

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

In order 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

In this section, we compare CRANE and FORCLIFT (Van den Broeck et al. 2011) on their ability to count various kinds of functions. (Other WFOMC algorithms such as 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 functions can be expressed in FOL. FORCLIFT then translates these sentences in FOL to formulas as defined in Definition 3.

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

$$\forall X \in a.\, \forall Y, Z \in b.\, p(X,Y) \wedge p(X,Z) \implies Y = Z$$

and

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

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

$$\forall W, X \in a.\, \forall Y \in b.\, p(W,Y) \wedge p(X,Y) \implies W = X$$

to restrict p to injections or

$$\forall Y \in b.\, \exists X \in a.\, p(X,Y)$$

to ensure surjectivity or remove Eq. (2) to consider partial functions. Lastly, one can replace all occurrences of b with a 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.[8] 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.

The experimental results are summarised in Table 1. 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 is also able to 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 $n \in \mathbb{N}_0$. In other words, the base case says that the empty partial map is the only partial injection with an empty domain, regardless of the codomain. 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 $j = 0, \ldots, n$.

[8] The search tree has a high branching factor, so exploring all nodes at depth 5 takes at most a few seconds whereas doing the same for depth 6 can be computationally infeasible in some cases.

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 b to a		
✓	✓		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 a, and n is the size of domain b. All asymptotic complexities are in $\Theta(\cdot)$. A dash means that no complete solution was found.

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 and a hybrid search algorithm. In Section 6, we saw examples of instances that become liftable not just in theory (Kuzelka 2021) but with an implemented algorithm as well. However, our experiments cover only a small set of instances since some parts of CRANE are yet to be fully automated. Specifically, our experiments focus on various function-counting problems—it remains to be seen what other types of instances become liftable as a result.

However, the most important direction for future work is in fully automating this new way of computing the (W)FOMC of a formula. 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 WMC. Second, these definitions must be simplified before they can be used, perhaps by a computer algebra system. Third, most importantly, we need a way to find the base cases for the recursive definitions provided by CRANE. What makes this problem non-trivial is that the number of base cases is not constant for functions of arity greater than one (i.e., formulas that mention more than one domain). On the other hand, assuming that a domain is empty can greatly simplify a problem. 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 make CRANE capable of automatically constructing efficient ways to compute a function (e.g., a sequence) of interest. In addition to the potential impact to 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.

Darwiche, A. 2020. Three Modern Roles for Logic in AI. In Suciu, D.; Tao, Y.; and Wei, Z., eds., *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2020, Portland, OR, USA, June 14-19, 2020*, 229–243. ACM.

Darwiche, A. 2021. Tractable Boolean and Arithmetic Circuits. In Pascal, H.; and Kamruzzaman, S. M., eds., *Neuro-Symbolic Artificial Intelligence: The State of the Art*, volume 342 of *Frontiers in Artificial Intelligence and Applications*, 146–172. IOS Press.

Darwiche, A. 2022. Causal Inference Using Tractable Circuits.

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.

Li, W.; Zeng, Z.; Vergari, A.; and Van den Broeck, G. 2021. Tractable computation of expected kernels. 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*, 1163–1173. AUAI Press.

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.

Shih, A.; and Ermon, S. 2020. Probabilistic Circuits for Variational Inference in Discrete Graphical Models. In Larochelle, H.; Ranzato, M.; Hadsell, R.; Balcan, M.; and Lin, H., eds., *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

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.; Meert, W.; and Davis, J. 2013. Lifted Generative Parameter Learning. In *Statistical Relational Artificial Intelligence, Papers from the 2013 AAAI Workshop, Bellevue, Washington, USA, July 15, 2013*, volume WS-13-16 of *AAAI Technical Report*. AAAI.

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.

Van Haaren, J.; Van den Broeck, G.; Meert, W.; and Davis, J. 2016. Lifted generative learning of Markov logic networks. *Mach. Learn.*, 103(1): 27–55.