Lifted Arbitrary Constraint Solving for Lifted Probabilistic Inference

Rodrigo de Salvo Braz
SRI International

Shahin Saadati
SRI International

Hung Bui
SRI International

Ciaran O'Reilly
SRI International

Abstract

Lifted Probabilistic Inference is a class of techniques for performing probabilistic inference on compact, intensionally defined relational graphical model representations without operating solely on the propositional level. This requires manipulating these representations and leaving them on the intensional level, where sets of parameterized random variables with the same marginal distributions are grouped together and manipulated all at once (and, as a result, more efficiently). A key problem in lifted inference is representing these groups compactly. This is done by associating constraints with their parameters, also called logic variables. [1] has shown that the ability to represent arbitrary groups can greatly increase lifted inference efficiency, even with constraint representation and algorithms with time dependent on domain size. We present the first constraint language and associated algorithms that allow arbitrary groupings with time complexity independent of the domain size (making it a lifted algorithm itself). This has the potential for even greater efficiency gains. Moreover, the flexibility of the framework and its ability to represent answers conditional on free variables allow for efficiency gains as well as much greater conceptual simplicity.

1 Introduction

Lifted Probabilistic Inference [2, 3, 4, 5, 6] is probabilistic inference on compact, intensionally defined relational graphical model representations without operating solely on the propositional level. It makes use of constraints on the parameters of parameterized random variable in order to keep the model compact and inference more efficient. We call the set of parameter value tuples a constraint represents its *extension*. To date, the representations used in lifted inference are either limited in their expressivity (that is, some extensions are not representable by a single constraint) [2, 3, 4, 5, 6], or can represent arbitrary extensions but have manipulation time dependent on the domain size (as in [1], who shows that arbitrary extensions allow greater efficiency even at greater manipulation costs).

We propose to represent these constraints as a formula in quantified, function-free *equality logic*, a logic with equality the only predicate. We then reduce the lifted inference operations on these constraints to two problems in this language: grounding size counting to *model counting with free variables*, and projection to *quantifier elimination with free variables*, and provide domain size-independent algorithms for them. This combination is the first constraint framework for lifted inference that can represent arbitrary extensions and the manipulation of which does not depend on domain size.

Model counting and quantifier elimination are problems of interest in their own right, not only in the context of lifted inference. In model counting, we compute the number of assignments to a given set of variables in the formula (called *indices*), but allow the presence of free variables in it as well, with a fixed but unknown value; thus, the number of models depends on these values. An example of this type of problem is to compute the number of tuples (x, y) where x, y belong to some fixed finite type of size T such that

$$x = A \wedge y \neq z \wedge x \neq y$$

where x, y are the index variables, A is a constant, and z is a free variable. It is easy to see that the number of solutions for the above constraints is $T - 1$ if $z = A$, and is $T - 2$ if $z \neq A$. Our goal is to compute this final expression given a formula and its indices, and to do so with complexity independent of T.

Quantifier elimination is the task of determining a non-quantified formula equivalent to a quantified one. For example, consider the formula $\exists x(x \neq y \wedge y = A)$, where the types of x and y are the same, and of size 10. Quantifier elimination gives us the equivalent formula $y = A$. It can be used to compute projections. For example, the projection of a tuple set $\{(x, y, z) : x \neq y \wedge y \neq z\}$ into (x, y) is described by $\{(x, y) : \exists z\ x \neq y \wedge y \neq z\}$, or more simply by the equivalent quantifier-free $\{(x, y) : x \neq y\}$.

This paper is organized as follows: first, we explain the link between lifted inference constraints, model counting, and quantifier elimination. After that, we define the syntax and semantics of our language. It is important to note that we define a super language of equality logic for algorithm description purposes, but solve model counting and quantifier eliminalition only on equality logic formulas. We then describe model counting by starting with the most basic cases, and generalizing, until an algorithm for full equality logic formulas is described. We then use this algorithm to define the quantifier elimination algorithm, followed by related work comments and conclusion.

2 Basic Definitions

2.1 Lifted Inference and Necessary Operations on Equality Logic

We assume reader familiarity with lifted inference and do not provide detailed descriptions of its representation and algorithms. We briefly outline the link between lifted inference and operations on equality logic.

The basic units in lifted inference are *parfactors* and *parameterized random variables*. Each parfactor and parameterized random variable has a *grounding*, which is the set of regular propositional factors and random variables they represent, respectively. Their representation involves a template factor or random variable with parameters, and a constraint determining the set of value tuples for these parameters. The grounding is defined as the set of factors or random variables obtained by the substitution of accepted parameter value tuples in the templates. We propose to represent this constraint as a formula in quantified, function-free equality logic, a language rich enough to represent any possible grounding, that is, what [1] call *arbitrary constraints*.

Lifted inference requires three key operations on constraints: *counting*, *splitting*, and *normalization*. Counting refers to determining the size of a unit's grounding in terms of a subset of its parameters; splitting refers to replacing units by sets of them with the same overall extensions, but respecting the property that each pair of parameterized random variables have either coinciding or mutually exclusive extensions; finally, normalization is the task of replacing a unit by a set of units, each of them with a constraint that is count-normalized with respect to a subset of parameters, that is, with a well-defined size for each possible assignment to the remaining parameters.

We reduce these operations to two problems on our Boolean formula language: grounding size counting to *model counting with free variables*, and splitting to *quantifier elimination with free variables*. It is not necessary to solve normalization separately anymore, because the two problems and their solutions accept free variables and provide answers conditioned on those free variables. In other words, normalization is integrated with the other two operations, a method which is both conceptually simpler and more efficient.

We illustrate the reduction to model counting with the following example. A common calculation performed by lifted inference algorithms is of the sort:

$$\prod_{x,y,z,w:x\neq w\wedge y\neq x\wedge y\neq z} \phi(p(x,z,w)),$$

where ϕ is an arbitrary non-negative real function on boolean-valued function p. Note that the variable y does not occur in $p(x,z,w)$, so for every assignment to x,z,w, the same value $\phi(p(x,z,w))$ is multiplied as many times as there are values of y compatible with x,z,w. This allows us to exponentiate it by this number and keep a more compact representation of the product:

$$= \prod_{x,z,w} \phi(p(x,z,w))^{|x\neq w\wedge y\neq x\wedge y\neq z|_y},$$

where $|x\neq w\wedge y\neq x\wedge y\neq z|_y$ is the number of values (models) of y satisfying $x\neq w\wedge y\neq x\wedge y\neq z$. Note that the remaining variables in the formula are free variables (in the scope of the model counting problem) and the number of models may depend on their values. Because, as we will see, our model counting algorithm provides conditional solutions, we obtain (assuming 10 possible values for each variable):

$$\begin{aligned}
&= \prod_{x,z,w} \phi(p(x,z,w))^{(\text{if } x\neq w \text{ then if } x=z \text{ then } 9 \text{ else } 8 \text{ else } 0)}\\
&= \Big(\prod_{x,z,w:x\neq w} \phi(p(x,z,w))^{(\text{if } x=z \text{ then } 9 \text{ else } 8)}\Big)\\
&\quad \Big(\prod_{x,z,w:x=w} \phi(p(x,z,w))^{0}\Big)\\
&= \prod_{x,z,w:x\neq w} \phi(p(x,z,w))^{(\text{if } x=z \text{ then } 9 \text{ else } 8)}\\
&= \Big(\prod_{x,z,w:x\neq w\wedge x=z} \phi(p(x,z,w))^{9}\Big)\\
&\quad \Big(\prod_{x,z:x\neq w\wedge x\neq z} \phi(p(x,z,w))^{8}\Big)\\
&= \Big(\prod_{x,w:x\neq w} \phi(p(x,x,w))^{9}\Big)\Big(\prod_{x,z,w:x\neq w\wedge x\neq z} \phi(p(x,z,w))^{8}\Big)\\
&= \Big(\prod_{x,w:x\neq w} \phi_3(p(x,x,w))\Big)\Big(\prod_{x,z,w:x\neq w\wedge x\neq z} \phi_4(p(x,z,w))\Big)
\end{aligned}$$

Note that there is no need for count normalization (with splitting of cases) of the constraints prior to counting. The conditions that *actually* influence the count are returned as part of the solution.

Another important lifted inference operation on constraints is splitting, which requires computing the instances of a parameterized random variable that occur in the grounding of a parfactor. For example, to compute the set of groundings for $p(x,y)$ occurring in the grounding

of a parfactor on $(p(z, A), p(A, w)), z \neq B$, we need to compute whether there are values of z, w such that $z \neq B \wedge ((x, y) = (z, A) \vee (x, y) = (A, w))$ or, in equality logic,

$$\exists z, w : z \neq B \wedge ((x = z \wedge y = A) \vee (x = A \wedge y = w))$$

which our quantifier elimination algorithm resolves to $(x \neq B \wedge y = A) \vee x = A$. Note that x, y are free variables, so dealing with them is essential, and the solution may depend on them, as in this case. Note also that the constraint for the parfactor intersection with $p(x, y)$ and its residual can be calculated as

$$z \neq B \wedge \exists x, y : (x = z \wedge y = A) \vee (x = A \wedge y = w)$$

and

$$z \neq B \wedge \neg\exists x, y : (x = z \wedge y = A) \vee (x = A \wedge y = w)$$

which is simple to do once we have an algorithm for solving arbitrary quantified equality logic formulas.

2.2 Language

We compute the number of models of formulas on equality. However our algorithms use a superset of the formula language, defined as follows.

2.2.1 Syntax

We assume a variable symbol set and a constant symbol set (including one for each natural number); constant symbols start with a capitalized letter and variable symbols with a small letter. *Formulas*, *Boolean* and *numerical expressions* and *expressions* are inductively defined as follows (note that any formula is a Boolean expression, and any Boolean or numeric expression is also an expression).

- a variable symbol is an **expression**;
- a (numerical/Boolean/other) constant symbol is a (numerical/Boolean/other) expression;
- the Boolean constants $False$ and $True$ are **formulas**;
- if α and β are variable or constant symbols of finite types, then $\alpha = \beta$ is a formula and, more specifically, an **atom**;
- if φ and φ' are formulas, then $\neg\varphi$, $\varphi \wedge \varphi'$, $\varphi \vee \varphi'$, $\varphi \Leftrightarrow \varphi'$, $\varphi \Rightarrow \varphi'$ are formulas; in particular, a **literal** is either an atom or its negation;
- if φ is a formula, then $\exists x\, \varphi$ and $\forall x\, \varphi$ are formulas;
- if α and β are expressions, then $\alpha = \beta$ is a Boolean expression;
- if α and β are numerical expressions, then $\alpha+\beta$, $\alpha\times\beta$, $\alpha - \beta$, $\alpha = \beta$ are numerical expressions;
- if $\alpha_1, \ldots, \alpha_n$ are expressions, then $\{\alpha_1, \ldots, \alpha_n\}$ is a **set** expression;
- if α is a set expression, then $|\alpha|$ is a numerical expression;
- if φ is a formula, then $|\varphi|_{x_1,\ldots,x_n}$ is a numerical expression;
- if φ is a formula and α and β are expressions of the same type, then $\text{if } \varphi \text{ then } \alpha \text{ else } \beta$ is an expression of their type.

Note that not all numeric expressions are formulas. Those using equality as the only predicate and finite types are formulas, but others (using arithmetic operators) are not.

The expression (or formula) $\alpha \neq \beta$ stands for $\neg(\alpha = \beta)$. The expressions used to inductively form an expression as shown above are its **immediate sub-expressions**. **Sub-expressions** are defined by the transitive closure of immediate sub-expressions. **Sub-formulas** are formula sub-expressions.

2.2.2 Semantics

An **interpretation** is a function from an expression to an element in a domain.[1]

Given a base interpretation $\mathcal{I}_0$ from variable and constant symbols to the domain (which we assume to map numeric constants to the corresponding numbers and the constants $False, True$ to the corresponding Boolean values), we define an interpretation function $\mathcal{I}$ based on $\mathcal{I}_0$ in the following way (for lack of space, we only define the two less standard forms – the remaining ones are defined in the usual way):

- $\mathcal{I}(|\varphi|_{x_1,\ldots,x_n}) =$ number of interpretations $\mathcal{I}'$ agreeing with $\mathcal{I}$ on all symbols but $x_1, \ldots, x_n$ such that $\mathcal{I}'(\varphi)$ is true (several examples are shown in Section 2.3);
- $\mathcal{I}(\text{if } \varphi \text{ then } \alpha \text{ else } \beta) = \begin{cases} \mathcal{I}(\alpha) & \text{if } \mathcal{I}(\varphi) \text{ is true} \\ \mathcal{I}(\beta) & \text{otherwise.} \end{cases}$

In summary, we define several types of expressions, including Boolean formulas with quantifiers where all literals are either equalities or disequalities on a set of variables and constants (hence, function-free). There are multiple types, or sorts, including, but not restricted to, Booleans and integers.

We assume that variables are only compared to constants with interpretations in their own type, and to variables with the same type. Distinct constants are assumed to have distinct interpretations (this does not incur in loss of generality, because if we wish to treat a constant as possibly equal

[1] Our terminology for semantics follows that of higher-order logic more closely than that of propositional or first-order logic.

to other constants, we can simply represent it with a variable instead).

Besides formulas, we define other types of expressions, such as arithmetic function applications, extensionally defined sets, cardinality of sets, and an if-then-else expression of the form "if φ then α else β", which has the same interpretation as that of α if the formula φ is true in the current interpretation, and β otherwise.

A **free variable** in an expression is a variable not quantified in it by $\exists, \forall$ or $|\cdot|_{x_1,\ldots,x_n}$. The **substitution** $\varphi[x/\phi]$ of a symbol x in φ by an expression ϕ is the expression obtained by replacing free occurrences of x in φ by ϕ, after appropriately standardizing ϕ and φ apart.

Two expressions α and β are **equivalent** if for all interpretations $\mathcal{I}$, $\mathcal{I}(\alpha) = \mathcal{I}(\beta)$.

2.3 The Model Counting Problem

Our main problem is computing the number $|\varphi|_{x_1,\ldots,x_n}$ of solutions, or models, of a formula φ with respect to the **indices** $x_1, \ldots, x_n$ as a function of the values of its free variables. Given an interpretation $\mathcal{I}$, a **solution**, or **model** is an interpretation $\mathcal{I}$' agreeing with $\mathcal{I}$ on all symbols but the indices (that is, a choice of assignment for the indices) such that $\mathcal{I}'(\varphi)$ is true. We assume that each variable x has a **type** denoted $type(x)$ of known finite size.

The number of models of φ on an empty set of variables is denoted $|\varphi|_\emptyset$. Since there is only one interpretation of an empty set of indices (the empty interpretation), $|\varphi|_\emptyset$ is 1 if φ is true, and 0 otherwise.

A **counting-solution** is either a numeric constant expression, or an if-then-else expression "if φ then α else β" where φ is a formula and α and β are counting-solutions. The **model counting problem** consists of determining a counting-solution equivalent to $|\varphi|_{x_1,\ldots,x_n}$. The main contribution of this paper is an algorithm for the model counting problem.

In the following examples of model counting problems and their counting-solutions, we assume all variables have the same type of size 10.

#	Problem	Counting-solution
1	$\lvert x = A\rvert_x$	1
2	$\lvert x \neq A\rvert_x$	9
3	$\lvert x = A \wedge y = B\rvert_{x,y}$	1
4	$\lvert x = A \wedge y \neq B\rvert_{x,y}$	9
5	$\lvert x \neq A \wedge y \neq B\rvert_{x,y}$	81
6	$\lvert y = B\rvert_{x,y}$	10
7	$\lvert y \neq B\rvert_{x,y}$	90
8	$\lvert True\rvert_{x,y}$	100
9	$\lvert False\rvert_{x,y}$	0
10	$\lvert x \neq A \wedge x \neq z\rvert_x$	if $z = A$ then 9 else 8
11	$\lvert z = C \wedge y \neq B\rvert_{x,y}$	if $z = C$ then 90 else 0
12	$\lvert z = C\rvert_\emptyset$	if $z = C$ then 1 else 0
13	$\lvert x = A\rvert_\emptyset$	if $x = A$ then 1 else 0
14	$\lvert\exists x(x = y \wedge z = A)\rvert_y$	if $z = A$ then 10 else 0

Note how z is a free variable in all problems in which it appears, because it never shows in the indices. The numbers of models in the indices of Problems 11 and 12 depend on the interpretation of z; if that is not C, then the formula is false and the number of models is 0. x is an index in Problems 1 to 11, and a free variable in Problem 13 (making the solution dependent on its interpretation).

3 An Algorithm for the Model Counting Problem

Algorithm 2 computes the counting-solution given a model counting problem. This section explains it and delineates its correctness proof.

We start by first discussing the important notion of *simplification*, then indicating how to compute the cardinality of an extensionally defined set (that is, of the form $\{\alpha_1, \ldots, \alpha_n\}$), and finally showing how to solve increasingly general cases of the model counting problem by reducing them to simpler ones.

3.1 Simplification

Throughout the algorithms, whenever a new expression is formed it is *simplified*. Most of the purpose of simplification is just to keep expressions small, but it also has the goal of keeping expressions in a normal form expected by the algorithms. We present simplification steps below, and comment on such normalization afterwards.

Simplification involves exhaustively applying the following steps until no further changes are possible:

1. if-then-elses are *externalized*, that is, expressions of the form

   $$f(t_1, \ldots, t_{i-1}, (\text{if } \varphi \text{ then } \alpha \text{ else } \beta), t_{i+1}, \ldots, t_n)$$

   with f any function other than if-then-else, are replaced by

   $$\begin{aligned}&\text{if } \varphi \text{ then } f(t_1, \ldots, t_{i-1}, \alpha, t_{i+1}, \ldots, t_n)\\ &\quad\text{else } f(t_1, \ldots, t_{i-1}, \beta, t_{i+1}, \ldots, t_n)\end{aligned}$$

   For example, $1 + (\text{if } x = A \text{ then } 2 \text{ else } 3)$ is replaced by if $x = A$ then $1 + 2$ else $1 + 3$.

2. Operations on constants are performed and replaced by the result (for example, $2 + 2$ is replaced by 4, $0 \neq 1$ is replaced by $True$, $A = B$ is replaced by $False$, $\neg False$ is replaced by $True$, $False \wedge True$ is replaced by $False$, if $True$ then 1 else 2 is replaced by 1 etc.).

3. Operations whose results can be defined by a subset of their arguments equal to certain constants ($\vee, \wedge, \Rightarrow, \Leftrightarrow, +, \times, -$) are replaced by their results. This corresponds to the notion of short-circuiting. For example, $False \wedge x = y$ is replaced by $False$ and $0 \times |x \neq z|_x$ is replaced by 0, if $True$ then $|x = A|_x$ else 2 is replaced by $|x = A|_x$, if $x \neq A$ then $True$ else $True$ is replaced by $True$ etc.

4. Equalities $\alpha = \alpha$ are replaced by $True$. As a consequence, disequalities $\alpha \neq \alpha$ are replaced by $False$.

5. Conjunctions including conjuncts $x = \alpha$ and $x = \beta$, where α, β are two distinct constants, are replaced by $False$. The DeMorgan's analog, disjunction of $x \neq \alpha$ and $x \neq \beta$, is replaced by $True$.

6. Conjunctions including conjuncts $x = \alpha$ and $x \neq \alpha$ where α is either a constant or variable are replaced by $False$. The DeMorgan's analog, disjunction of $x \neq \alpha$ and $x = \alpha$, is replaced by $True$.

7. Conjunctions including conjuncts $x = \alpha$ and $x \neq \beta$, where α, β are two distinct constants, are replaced by $x = \alpha$. Disjunctions including disjuncts $x = \alpha$ and $x \neq \beta$ are replaced by $x \neq \beta$.

It is possible to show that simplification halts and produces an expression equivalent to the initial one. Moreover, the simplified expression exhibits a couple of normal properties assumed by the algorithm's operations. The first one is that if-then-else operations are *externalized*, that is, that no operations other than if-then-elses contain if-then-elses as arguments. This ensures that counting-solutions are either numbers or if-then-elses with counting-solutions as their then and else branches. Another expected property is that conjunctive clauses (conjunctions of literals) do not contain equalities or disequalities of the type $x = x$ or $x \neq x$. We remark on the need for these properties at the relevant point of the explanation of the algorithm.

3.2 Cardinality of extensionally defined sets

A basic problem we reduce the model counting problem to is the computation of the cardinality of extensionally defined sets, that is, sets represented in the form $\{\alpha_1, \ldots, \alpha_n\}$, where each α_i is either a variable or a constant. This is done by Algorithm 1 by considering whether each element α is equal to any of the subsequent ones. If so, α is redundant and the answer is equal to the cardinality of the remainder of the set. If α is distinct from all other elements, the answer is equal to the cardinality of the remainder of the set plus 1, since α will be an extra element.

$$
\begin{aligned}
|\{A, B, C\}| &= 3 \\
|\{x, y\}| &= \text{if } x = y \text{ then } 1 \text{ else } 2 \\
|\{x, y, C\}| &= \text{if } x = y \vee x = C \\
&\qquad \text{then if } y = C \text{ then } 1 \text{ else } 2 \\
&\qquad \text{else if } y = C \text{ then } 2 \text{ else } 3
\end{aligned}
$$

Now we show how to solve increasingly general cases of the model counting problem by reducing it to simpler cases.

3.3 Counting solutions of a formula with no indices

Let us consider solving $|\varphi|_\emptyset$. There is only one interpretation to the empty set of indices (the empty interpretation).

Algorithm 1 Compute the cardinality of an extensionally defined set.

Require: $S = \{t_1, \ldots, t_n\}$, possibly the empty set. Each t_i is a constant or a free variable.

```
1: function COMPUTECARDINALITY(S)
2:     if S = {} then
3:         return 0
4:     else
5:         α ← computeCardinality({t_2, ..., t_n})
6:         return simplify(if t_1 = t_2 ∨ ··· ∨ t_1 = t_n then α else α + 1)
7:     end if
8: end function
```

This interpretation does not change φ when applied to it, and therefore will be a model if φ is true, and not be a model if φ is false. That is to say, $|\varphi|_\emptyset = \text{if } \varphi \text{ then } 1 \text{ else } 0$. For example,

$$|x \neq y \wedge y \neq A|_\emptyset \;=\; \text{if } x \neq y \wedge y \neq A \text{ then } 1 \text{ else } 0$$

3.4 Counting solutions of a formula with one index

We now consider how to compute $|\varphi|_x$ for a formula φ and single index x.

If φ is a conjunctive clause of disequalities with a single index (including the case with no disequalities at all, that is, $True$), the number of models for the index is equal to its type cardinality minus the cardinality of the extensionally defined set containing the values it is constrained to be distinct from. In the following, we assume $|type(x)| = 10$:

$$
\begin{aligned}
&|x \neq y \wedge x \neq z \wedge x \neq C|_x \\
&= |type(x)| - |\{y, z, C\}| \quad \text{(assuming } \{y, z, C\} \subseteq type(x)\text{)} \\
&= 10 - \text{if } y = z \vee y = C \\
&\qquad \text{then if } z = C \text{ then } 1 \text{ else } 2 \\
&\qquad \text{else if } z = C \text{ then } 2 \text{ else } 3 \\
&= \text{if } y = z \vee y = C \\
&\qquad \text{then if } z = C \text{ then } 9 \text{ else } 8 \\
&\qquad \text{else if } z = C \text{ then } 8 \text{ else } 7
\end{aligned}
$$

Note that the correctness of this transformation depends on the conjunctive clause being in normal form and therefore not containing disequalities of the form $x \neq x$. If it did, we would rewrite $|x \neq x \wedge x \neq A|_x$ as $|type(x)| - |\{x, A\}|$, which would incorrectly have x as a free variable.

If φ is a conjunctive clause including equalities on the index, we replace the index by its value and remove it from the index list.

$$
\begin{aligned}
&|x = A \wedge x \neq y \wedge x \neq z \wedge x \neq C|_x \\
&= |A \neq y \wedge A \neq z \wedge A \neq C|_\emptyset \\
&= |y \neq A \wedge z \neq A|_\emptyset \\
&= \text{if } y \neq A \wedge z \neq A \text{ then } 1 \text{ else } 0
\end{aligned}
$$

Note that the correctness of this transformation depends on the conjunctive clause being in normal form and therefore not containing equalities of the form $x = x$. If it did, we would rewrite $|x = x \wedge x \neq A|_x$ as $|x \neq A|_\emptyset$, which would incorrectly remove x as an index without eliminating it from the formula.

If φ is a conjunctive clause including literals that do not involve the index, we remove them from the formula and condition the number of models on them:

$$\begin{aligned}
&|z \neq y \wedge y \neq B \wedge x \neq y \wedge x \neq z|_x \\
&= \text{if } z \neq y \wedge y \neq B \text{ then } |x \neq y \wedge x \neq z|_x \text{ else } 0 \\
&= \text{if } z \neq y \wedge y \neq B \\
&\quad \text{then if } y = z \text{ then } |type(x)| - 1 \text{ else } |type(x)| - 2 \\
&\quad \text{else } 0
\end{aligned}$$

If φ is a DNF (Disjunctive Normal Form, or a disjunction of conjunctive clauses), it is either an empty disjunction ($False$), or a disjunction of conjunctive clauses of the form $|\varphi_1 \vee \varphi_2 \vee \cdots \vee \varphi_n|_x$. If φ is $False$, the number of models is 0. If it is in the latter form, we use the the fact that $|\varphi_1 \vee \varphi_2|_x = |\varphi_1|_x + |\varphi_2|_x - |\varphi_1 \wedge \varphi_2|_x$[2], which in the DNF case translates to

$$\begin{aligned}
&|\varphi_1 \vee \varphi_2 \vee \cdots \vee \varphi_n|_x \\
&= |\varphi_1|_x + |\varphi_2 \vee \cdots \vee \varphi_n|_x - |\varphi_1 \wedge (\varphi_2 \vee \cdots \vee \varphi_n)|_x \\
&= |\varphi_1|_x + |\varphi_2 \vee \cdots \vee \varphi_n|_x - |(\varphi_1 \wedge \varphi_2) \vee \cdots \vee (\varphi_1 \wedge \varphi_n)|_x
\end{aligned}$$

which is correct by induction on the number of disjuncts, with the computation on conjunctive clauses being the base case.

If φ is a quantifier-free formula but not a DNF, we compute a DNF φ' equivalent to it and return $|\varphi'|_x$.[3]

If φ is a formula containing quantifiers, we compute a quantifier-free formula φ' equivalent to it and return $|\varphi'|_x$. We show how to eliminate quantifiers in Section 4.

3.5 Counting solutions of a formula with multiple indices

We now consider the computation of $|\varphi|_{x_1,\ldots,x_n}$ for multiple indices $x_1, \ldots, x_n$.

[2]This follows from the fact that, for any two sets A and B, $|A \cup B| = |A| + |B| - |A \cap B|$, applied to the sets of solutions of each formula, and from the fact that the set of solutions of a disjunction is the union of the sets of solutions of the disjuncts

[3]It is very important to note that, in practice, one should not implement this algorithm by naively converting the entire formula to a DNF, because it may be that solving only one sub-formula of it is enough to solve the entire problem, in which case converting the remaining sub-formulas to DNF would have been a waste. One can clearly see this if the input is $x \neq A \wedge x = A \wedge \varphi$ where φ is a large formula; clearly one does not need to consider φ at all in order to decide that the formula is equivalent to $False$. It is much more efficient to only convert parts of the formula to DNF on the fly, as they are needed for the procedure, in a technique known as *lazy case-splitting* [7]. This technique is largely orthogonal to our contribution, so we do not detail it here.

Before we do that, however, we show how to obtain counting-solutions equivalent to summations of the type

$$\sum_{x:\varphi} S$$

where φ is a formula constraining the value of the summation index x and S is a counting-solution. The summation is over the values of x satisfying formula φ. We show how to do this *without* iterating over all possible values of x, but instead in time *constant* in its type size. If S is a number N, we simply have

$$\sum_{x:\varphi} N = |\varphi|_x \times N$$

which is simplified to a number, which is a counting-solution. If S is of the form $\text{if } \varphi' \text{ then } S_1 \text{ else } S_2$, where φ' is a formula and S_1, S_2 are counting-solutions, we have

$$\sum_{x:\varphi} \text{if } \varphi' \text{ then } S_1 \text{ else } S_2 = \Big(\sum_{x:\varphi \wedge \varphi'} S_1\Big) + \Big(\sum_{x:\varphi \wedge \neg\varphi'} S_2\Big)$$

with two summations on strictly smaller counting-solutions that can be solved recursively. The simplification of an addition of two counting-solutions results in a single counting solution.

Now that we know how to solve such summations, we can solve model counting problems with more than one index by reducing them to a problem with one less index. Let x be an index and $\mathbf{y}$ be a set of indices. Then we have

$$|\varphi|_{x,\mathbf{y}} = \sum_x |\varphi|_{\mathbf{y}}$$

because the set of solutions for $x, \mathbf{y}$ is the union of the sets of counting-solutions for $\mathbf{y}$ for each value of x, and those sets are mutually exclusive because the solutions in each of them have a distinct value for x. We can then solve the summation as shown before.

Let us see a complete example, assuming $|type(x)| = |type(y)| = 10$:

$$\begin{aligned}
&|y \neq x \wedge y \neq A|_{x,y} \\
&= \sum_x |y \neq x \wedge y \neq A|_y \\
&= \sum_x \text{if } x = A \text{ then } |type(y)| - 1 \text{ else } |type(y)| - 2 \\
&= \sum_x \text{if } x = A \text{ then } 9 \text{ else } 8 \\
&= \Big(\sum_{x:x=A} 9\Big) + \Big(\sum_{x:x\neq A} 8\Big) \\
&= |x = A|_x \times 9 + |x \neq A|_x \times 8 \\
&= 1 \times 9 + (|type(x)| - 1) \times 8 \\
&= 9 + 9 \times 8 \\
&= 81
\end{aligned}$$

Note how, in the subproblem indexed by y alone, x is a free variable. Even when a problem does not contain any free variables (not the case here), it is reduced to one with free variables. Therefore, an algorithm for model counting for formulas with free variables is not only more useful because it solves a more general problem than the one without free variables, it also provides an elegant inductive reduction.

While the reduction to a summation shown above is enough to solve all problems with more than one index, a more efficient reduction applies when the formula is a conjunction with a conjunct being an equality on an index:

$$|x = A \wedge (y \neq B \vee z \neq x)|_{x,y,z} = |y \neq B \vee z \neq A|_{y,z}$$

which is the same operation utilized to solve conjunctive clauses with index equalities for the one-index case.

4 A Quantifier Elimination Algorithm

The previous section describes how to solve model counting of quantifier-free formulas. When a formula has quantifiers, first we eliminate them, finding an equivalent quantifier-free formula, and then compute its (same) number of models. Therefore, we need to describe how to eliminate quantifiers from a formula φ. This is a problem in itself and useful even in contexts other than model counting (for example, it can be used for computing a *projection* in relational algebra).

The method is described by Algorithm 3. If φ is an existential or universal quantification, we remove the top quantifier by reducing the problem to a model counting problem. Using quantifier elimination for solving model counting, and model counting for solving quantifier elimination may seem circular at first, but is correct by induction on the formula size.

If φ is an existential quantification $\exists x\, \varphi'$, it is equivalent to the simplification of $|\varphi'|_x \neq 0$. For example:

$$\begin{aligned} \exists x\, x = A \wedge x \neq A \\ \Leftrightarrow \quad & |x = A \wedge x \neq A|_x \neq 0 \\ \Leftrightarrow \quad & 0 \neq 0 \\ \Leftrightarrow \quad & False \end{aligned}$$

The comparison of an if-then-else counting solution "if ψ then S_1 else S_2" to 0 is reduced to a comparison between numbers by if-then-else externalization, as seen in the following example where $|type(x)| = 2$:

$$\begin{aligned} \exists x\, x \neq A \wedge x \neq z \;\Leftrightarrow\; & |x \neq A \wedge x \neq z|_x \neq 0 \\ \Leftrightarrow\; & 2 - (\text{if } z = A \text{ then } 1 \text{ else } 2) \neq 0 \\ \Leftrightarrow\; & (\text{if } z = A \text{ then } 2 - 1 \text{ else } 2 - 2) \neq 0 \\ \Leftrightarrow\; & (\text{if } z = A \text{ then } 1 \text{ else } 0) \neq 0 \\ \Leftrightarrow\; & \text{if } z = A \text{ then } 1 \neq 0 \text{ else } 0 \neq 0 \\ \Leftrightarrow\; & \text{if } z = A \text{ then } True \text{ else } False \\ \Leftrightarrow\; & z = A \end{aligned}$$

Algorithm 2 Computes a counting-solution for a given formula. Assumes that simplification and conversion of quantifier-free formulas to DNFs are already defined elsewhere.

function COUNT($\varphi, indices$)
  if $indices = \{\}$ **then**
    return simplify(if φ then 1 else 0)
  else if $indices = \{x\}$ **then** ▷ indices has exactly one element
    return countOfDNF(convertToDNF(φ), x)
  else ▷ indices are $\{x_1, \ldots, x_n\}$, $n > 1$
    return computeSum($x_1, True$, count($\varphi, \{x_2, \ldots, x_n\}$))
  end if
end function
function CONVERTTODNF(φ)
  return quantifierFreeFormToDNF(eliminateQuantifiers(φ))
end function
function COUNTOFDNF(φ, x)
  if φ is $False$ **then**
    return 0
  else if φ has the form $\psi_1 \vee \psi_2$ **then**
    return count(ψ_1, x) + count(ψ_2, x) − count($\psi_1 \wedge \psi_2, \{x\}$)
  else ▷ φ is a conjunctive clause
    return countOfConjunctiveClause(φ, x)
  end if
end function
function COUNTOFCONJUNCTIVECLAUSE(φ, x)
  $\varphi_1 \leftarrow$ conjunction of all conjuncts of φ containing x (possibly empty)
  $\varphi_2 \leftarrow$ conjunction of all conjuncts of φ *not* containing x (possibly empty)
  if φ_1 has the form $\psi \wedge (x = t)$ **then** ▷ ψ may just be True
    $\alpha \leftarrow$ count($\psi[x/t], \emptyset$)
  else ▷ φ_1 is $x \neq t_1 \wedge \cdots \wedge x \neq t_k$ (possibly empty)
    $\alpha \leftarrow$ simplify($|type(x)|$ − computeCardinality($\{t_1, \ldots, t_k\}$))
  end if
  return simplify(if φ_2 then α else 0)
end function
function COMPUTESUM(x, ψ, α) ▷ computes $\sum_{x:\psi} \alpha$, where α is a counting-solution
  if α has the form if φ then t_1 else t_2 **then**
    return simplify(computeSum($x, \psi \wedge \varphi, t_1$) + computeSum($x, \psi \wedge \neg\varphi, t_2$))
  else ▷ α is a number
    return simplify(count($\psi, \{x\}$) × α)
  end if
end function

We eliminate a universal quantifier in the analogous manner by using the equivalence $\forall x\, \varphi \Leftrightarrow |\varphi|_x = |type(x)|$. In the example below, $|type(x)|$ is 10.

$$\begin{aligned}\forall x\, x \neq A \wedge x \neq y \;&\Leftrightarrow\; |x \neq A \wedge x \neq y|_x = |type(x)| \\ &\Leftrightarrow\; (\text{if } y = A \text{ then } 9 \text{ else } 8) = 10 \\ &\Leftrightarrow\; \text{if } y = A \text{ then } 9 = 10 \text{ else } 8 = 10 \\ &\Leftrightarrow\; \text{if } y = A \text{ then } False \text{ else } False \\ &\Leftrightarrow\; False\end{aligned}$$

If φ is not an existential or universal quantification, it may be that its sub-expressions contain quantifiers. In this case we replace them by quantifier-free versions recursively computed. This is correct by induction on the expression size, with a base case for variable and constant symbols, which are already quantifier-free. For example:

$$x \neq A \wedge \exists y\, (y = B \wedge y \neq x) \quad \Leftrightarrow \quad x \neq A \wedge x \neq B$$

A very important observation regarding quantifier elimination is that, even though it can be reduced to a model counting problem, it is fundamentally easier than the latter, because it is easier to decide that a formula has a number of models distinct from 0 (or a type's size) than to decide its precise number of models. For example, if $|type(x)| > 3$,

$$|x \neq y \wedge x \neq z \wedge x \neq w|_x = |type(x)| - |\{y, z, w\}| \neq 0$$

can be decided without computing the extensionally defined set cardinality, since it has an upper bound of 3 and the subtraction will be greater than 0. Another example is the reduction for disjunctions. When trying to decide

$$|\varphi_1 \vee \varphi_2|_x = |\varphi_1|_x + |\varphi_2|_x - |\varphi_1 \wedge \varphi_2|_x \neq 0$$

it is only necessary to compute $|\varphi_1|_x$ and $|\varphi_2|_x$; if either of them is distinct from 0, so is the total, and if they are both 0, so are $|\varphi_1 \wedge \varphi_2|_x$ and, as a consequence, the total. In general, one can write a version of the model counting algorithm that takes an extra input indicating that we only need to decide if the number of models is distinct from a given number, and returns $True$ as soon as that can be determined, even before computing the exact counting solution.

5 Complexity

A complexity analysis shows that the model counting algorithm is $O(exp(exp(n)))$ for n the number of free variables and indices. The double exponential comes from the fact that counting the models of a disjunction with m disjuncts requires two recursive calls on disjunctions with $m-1$ disjuncts. When eliminating quantifiers, only one such call is needed and the complexity becomes $O(exp(n))$, the same of satisfiability.[4]

[4]Note that certain classes of constraints, for example those that contain an arbitrary half of elements in the domain, have a size dependent on the domain size, and in those cases the algorithm is dependent on the domain size. However, it remains true that the algorithm is independent of the domain size given the constraint size.

Algorithm 3 Computes a quantifier-free formula equivalent to a given formula.

```
1: function ELIMINATEQUANTIFIERS(φ)
2:     if φ has the form ∃x ψ then
3:         return simplify(count(ψ, {x}) ≠ 0)
4:     else if φ has the form ∀x ψ then
5:         return simplify(count(ψ, {x}) = |type(x)|)
6:     end if
7:     φ ← simplify(replace each sub-expression φ of φ
   by eliminateQuantifiers(φ))
8:     return φ
9: end function
```

6 Related Work

There has been work on the *satisfiability* of propositional logic with equalities [8, 9, 10, 11], but it has not been extended to computing the number of models. Propositional model counting [12, 13] (#SAT, an extension of SAT to also compute the number of satisfying models) is a special case of our problem where each proposition p can be represented by an index Boolean variable p in an equality $p = True$ and the formula contains no free variables or quantifiers. Our focus however is not just on Boolean variables, but on handling variables of arbitrarily large finite types.

Our model counting problem can be reduced to #SAT by creating a proposition to represent $x = A$ for each variable/value pair. However, this requires the encoding of $x = \alpha \Rightarrow x \neq \beta$ for every pair of constants α, β, thus depending on T. For this same reason, it seems unlikely that a reduction to a model counting problem on Boolean-valued variables can be found. We are not aware of an algorithm proposed in the literature for model counting that does not depend on the type size.

The closest contributions have been the ones proposed by the lifted probabilistic inference community, particularly [1, 6, 4]. Our contribution provides lifted inference algorithms with a rich language that can compactly represent arbitrary constraints, that is, any set of assignments to variables. It has been shown by [1] that this significantly improves lifted inference performance. In the lifted inference literature so far, only [1] can represent arbitrary constraints, but their space and time complexities depend on type size. Our solution is the first arbitrary constraint language with time and space complexity independent of type size. Another important advantage of our approach is the ability of returning solutions conditional on free variables, which does away with the need for normalizing constraints in advance.

7 Conclusion

We presented two algorithms: one that solves model counting for function-free, existentially and universally quantified Boolean formulas where the only predicate is equality, with free variables (in which case the answer depends on those variables), and another, naturally derived from the first one, that eliminates quantifiers from such formulas. They are implemented and available for downloading. Model counting is relevant to any application in which a set of objects is intensionally defined through a constraint of this type, and the number of such objects needs to be computed. In particular, it is relevant to lifted probabilistic inference, a generalization of inference for graphical models that can take more expressive and compact logic-like representations. Our solution is superior to previous solutions from that community in that it represents and manipulates arbitrary constraints in space and time independent of type size, and accepts quantifiers and free variables that are not required to be previously pairwise constrained for either equality or disequality. This makes its use both more convenient and more efficient in that context. Furthermore, it is an instance of symbolic evaluation, which makes it easier to be integrated into other algorithms also using symbolic evaluation. Future work includes empirical evaluation, generalizing it to richer languages, including uninterpreted functions, arithmetic operators on variables, and interpreted predicates.

8 Acknowledgments

The authors gratefully acknowledge the support of the Defense Advanced Research Projects Agency (DARPA) Machine Reading Program under Air Force Research Laboratory (AFRL) prime contract no. FA8750-09-C-0181. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the view of DARPA, AFRL, or the US government. Approved for Public Release, Distribution Unlimited.

References

[1] Taghipour, N., Fierens, D., Davis, J., Blockeel, H.: Lifted variable elimination with arbitrary constraints. Journal of Machine Learning Research - Proceedings Track **22** (2012) 1194–1202

[2] Poole, D.: First-order probabilistic inference. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence. (2003) 985–991

[3] de Salvo Braz, R., Amir, E., Roth, D.: Lifted first-order probabilistic inference. In: Proceedings of IJCAI-05, 19th International Joint Conference on Artificial Intelligence. (2005)

[4] Milch, B., Zettlemoyer, L., Kersting, K., Haimes, M., Kaelbling, L.P.: Lifted probabilistic inference with counting formulas. In: Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI-2008), Chicago, Illinois, USA (July 2008 2008)

[5] Singla, P., Domingos, P.: Lifted first-order belief propagation. In: Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI-2008), Chicago, Illinois, USA (July 2008 2008)

[6] Kisynski, J., Poole, D.: Constraint processing in lifted probabilistic inference. In: UAI. (2009)

[7] Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. 1 edn. Springer Publishing Company, Incorporated (2008)

[8] Meir, O., Strichman, O.: Yet another decision procedure for equality logic. In: Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings. (2005) 307–320

[9] Gammer, I., Amir, E.: Solving satisfiability in ground logic with equality by efficient conversion to propositional logic. In: Abstraction, Reformulation, and Approximation, 7th International Symposium, SARA 2007, Whistler, Canada, July 18-21, 2007, Proceedings. (2007) 169–183

[10] Tveretina, O.: Deciding satisfiability of equality logic formulas with uninterpreted functions. In: Joint Annual Workshop of ERCIM/CoLogNet on Constraint Solving and Constraint Logic Programming, Lausanne, Switzerland. (2004)

[11] Zantema, H., Groote, J.F.: Transforming equality logic to propositional logic. Electr. Notes Theor. Comput. Sci. **86**(1) (2003) 162–173

[12] Arora, S., Barak, B.: Computational Complexity - A Modern Approach. Cambridge University Press (2009)

[13] Roth, D.: On the hardness of approximate reasoning.
Artificial Intelligence **82**(1-2) (Apr 1996) 273–302