EasyChair Preprint

№ 210

Constraint-Based Inference in Probabilistic Logic Programs

Arun Nampally, Timothy Zhang and C. R. Ramakrishnan

June 1, 2018

Constraint-Based Inference in Probabilistic Logic Programs

Arun Nampally, Timothy Zhang, C. R. Ramakrishnan
Department of Computer Science, Stony Brook University, Stony Brook, NY 11794
{anampally,thzhang,cram}@cs.stonybrook.edu

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Probabilistic Logic Programs (PLPs) generalize traditional logic programs and allow the encoding of models combining logical structure and uncertainty. In PLP, inference is performed by summarizing the possible worlds which entail the query in a suitable data structure, and using this data structure to compute the answer probability. Systems such as ProbLog, PITA, etc., use propositional data structures like explanation graphs, BDDs, SDDs, etc., to represent the possible worlds. While this approach saves inference time due to substructure sharing, there are a number of problems where a more compact data structure is possible. We propose a data structure called Ordered Symbolic Derivation Diagram (OSDD) which captures the possible worlds by means of constraint formulas. We describe a program transformation technique to construct OSDDs via query evaluation, and give procedures to perform exact and approximate inference over OSDDs. Our approach has two key properties. Firstly, the exact inference procedure is a generalization of traditional inference, and results in speedup over the latter in certain settings. Secondly, the approximate technique is a generalization of likelihood weighting in Bayesian Networks, and allows us to perform sampling-based inference with lower rejection rate and variance. We evaluate the effectiveness of the proposed techniques through experiments on several problems.

1 Introduction

A wide variety of models that combine logical and statistical knowledge can be expressed succinctly in the Probabilistic Logic Programming (PLP) paradigm. The expressive power of PLP goes beyond that of traditional probabilistic graphical models (eg. Bayesian Networks (BNs) and Markov Networks (MNs)) as can be seen in the examples in Figs. 1 and 2. These examples are written in PRISM, a pioneering PLP language (Sato and Kameya 1997). The example in Fig. 1 encodes the probability distribution of a palindrome having a specific number of occurrences of a given character, the example in Fig. 2 encodes the probability that at least two people in a given set have the same birthday. Examples such as these and other models, such as reachability over graphs with probabilistic edge relations, illustrate how logical clauses can be used to specify models that go beyond what is possible in traditional probabilistic graphical models.

The Driving Problem. The expressiveness of PLP comes at a cost. Since PLP is an extension to traditional logic programming, inference in PLP is undecidable in general. Inference is intractable even under strong finiteness assumptions. For instance, consider the PRISM program in Fig. 1. In that program, `genlist/2` defines a list of the outcomes of N identically distributed random variables ranging over {`a`,`b`} (through `msw/3` predicates). Predicate `palindrome/1` tests, using a definite clause grammar definition, if a given list is a palindrome; and `count_as/2` tests

```
% Generate a list of N random variables.
genlist(0, []).
genlist(N, L) :-
    N > 0,
    msw(flip, N, X),
    L = [X|L1],
    N1 is N-1,
    genlist(N1, L1).
% Evidence: list is a palindrome.
evidence(N) :-
    genlist(N, L), palindrome(L).
% Query: string has K 'a's
query(N, K) :-
    genlist(N, L), count_as(L, K).
% Check if a given list is a palindrome
palindrome(L) :- phrase(palindrome, L).
palindrome --> [].
palindrome --> [_X].
palindrome --> [X], palindrome, [X].
% Query condition:
count_as([], 0).
count_as([X|Xs], K) :-
    count_as(Xs, L),
    (X=a -> K is L+1; K=L).
% Domains:
values(flip, [a,b]).
% Distribution parameters:
set_sw(flip, [0.5, 0.5]).
```

Fig. 1. Palindrome PLP

if a given list contains k (not necessarily consecutive) "a"s. Using these predicates, consider the inference of the conditional probability of $\texttt{query}(n,k)$ given $\texttt{evidence}(n)$: i.e., the probability that an n-element palindrome has k "a"s.

The conditional probability is well-defined according to PRISM's distribution semantics (Sato and Kameya 2001). However, *PRISM itself will be unable to correctly compute the conditional query's probability* since the conditional query, as encoded above, will violate the PRISM system's assumptions of independence among random variables used in an explanation. Moreover, while the probability of goal `evidence(N)` can be computed in linear time (using explanation graphs), the conditional query itself is intractable since the computation is dominated by the binomial coefficient $\binom{N}{k}$. This is not surprising since probabilistic inference is intractable over even simple statistical models such as Bayesian networks. Consequently, exact inference techniques used in PLP systems such as PRISM, ProbLog (Raedt et al. 2007) and PITA (Riguzzi and Swift 2011), have exponential time complexity when used on such programs.

```
% Two from a population of
% size N share a birthday.
same_birthday(N) :-
    person(N, P1),
    msw(b, P1, D),
    person(N, P2),
    P1 < P2,
    msw(b, P2, D)
% Bind P, backtracking
% through 1..N
person(N, P) :-
    basics:for(P, 1, N).
% Distribution parameters:
:- set_sw(b(_),
    uniform(1,365)).
```

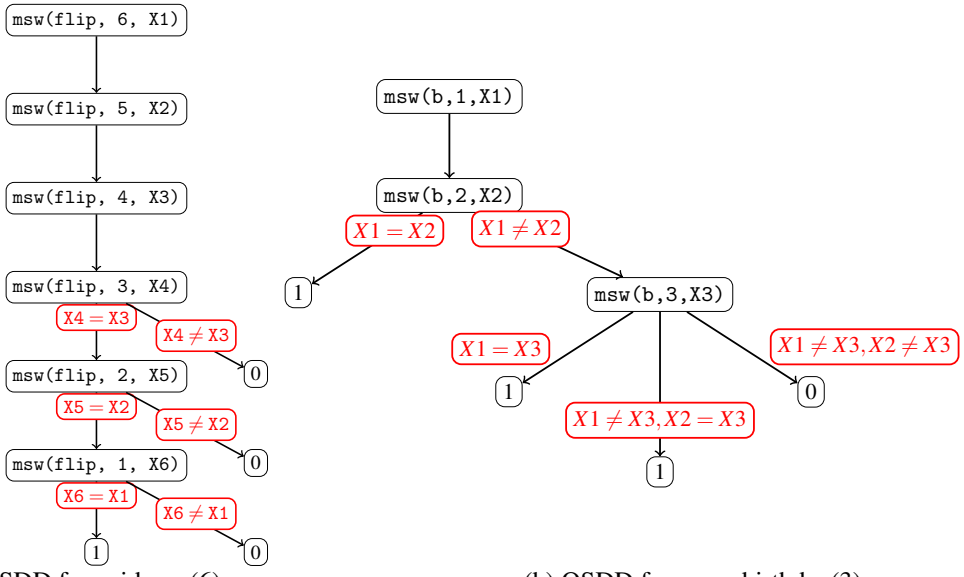
Fig. 2. Birthday PLP

Approximate inference based on rejection sampling also performs poorly, rejecting a vast number of generated samples, since the likelihood of a string being a palindrome decreases exponentially in N. Alternatives such as Metropolis-Hastings-based Markov Chain Monte Carlo (MCMC) techniques (Hastings 1970, e.g.) do not behave much better: the chains exhibit poor convergence (mixing), since most transitions lead to strings inconsistent with evidence. Gibbs-sampling-based MCMC (Geman and Geman 1984)

cannot be readily applied since the dependencies between random variables are hidden in the program and not explicit in the model.

Our Approach. In this paper, we use PRISM's syntax and distribution semantics, *but without the requirements imposed by the PRISM system*, namely, that distinct explanations of an answer are pairwise mutually exclusive and all random variables within an explanation are independent. We, however, retain the assumption that distinct random variable instances are independent. Thus we consider PRISM programs with their *intended* model-theoretic semantics, rather than that which is computed by the PRISM system.

We propose a data structure called *Ordered Symbolic Derivation Diagram* (OSDD) which represents the set of possible explanations for answers to a goal symbolically. The key characteristic of OSDDs is the use of constraints on random variables. This data structure is constructed through tabled evaluation on a transformed input program. For example, the OSDD for answers to goal "`same_birthday(3)`" from example Fig. 2 is shown in Fig. 3(b). This data structure can be used for performing exact inference in polynomial time as will be described later. In cases where exact inference is intractable, OSDDs can be used to perform sampling based inference. For example, the OSDD for answers to goal "`evidence(6)`" from Fig. 1 is shown in Fig. 3 (a) and can be used for performing likelihood weighted sampling (Fung and Chang 1990; Shachter and Peot 1990).



(a) OSDD for evidence(6) (b) OSDD for same_birthday(3)

Fig. 3. OSDDs for introductory examples

The rest of the paper is organized as follows. In Section 2 we formally define OSDDs and the operations on OSDDs. Next we give the procedure for construction of OSDDs. This procedure relies on a program transformation which is explained in Section 3. Next we give the exact and approximate inference algorithms using OSDDs in Section 4. We present the experimental results in Section 5, related work in Section 6, and concluding remarks in Section 7.

2 Ordered Symbolic Derivation Diagrams

PRISM. Programs in PRISM follow Prolog's syntax, as illustrated in Figs. 1 and 2. In a PRISM program, the `msw` relation ("multi-valued switch") has a special meaning: `msw(X,I,V)` says that `V` is the outcome of the `I`-th instance from a family `X` of random processes. The set of variables $\{\mathtt{V}_i \mid \mathtt{msw}(p,i,\mathtt{V}_i)\}$ are i.i.d. for a given random process p.

The distribution parameters of the random variables are specified separately.

The meaning of a PRISM program is given in terms of a *distribution semantics* (Sato and Kameya 1997; Sato and Kameya 2001). A PRISM program is treated as a non-probabilistic logic program over a set of probabilistic facts, i.e. the `msw` relation. An instance of the `msw` relation defines one choice of values of all random variables. A negation-free PRISM program is associated with a set of least models, one for each `msw` relation instance. A probability distribution is then defined over the set of models, based on the probability distribution of the `msw` relation instances. This distribution of least models is the semantics of a PRISM program. Note that the distribution semantics is declarative. For a subclass of programs, PRISM has an efficient procedure for computing this semantics based on OLDT resolution (Tamaki and Sato 1986).

Inference in PRISM proceeds as follows. When the goal selected at a step is of the form `msw(X,I,Y)`, then `Y` is bound to a possible outcome of a random process `X`. *Thus in PRISM, derivations are constructed by enumerating the possible outcomes of each random variable.* The derivation step is associated with the probability of this outcome. If all random processes encountered in a derivation are independent, then the probability of the derivation is the product of probabilities of each step in the derivation. If a set of derivations are pairwise mutually exclusive, the probability of the set is the sum of probabilities of each derivation in the set. PRISM's evaluation procedure is defined only when the independence and exclusiveness assumptions hold. Finally, the probability of an answer is the probability of the set of derivations of that answer.

Notation. We assume familiarity with common logic programming terminology such as variables, terms, substitutions, predicates and clauses. We use Prolog's convention, using identifiers beginning with lower case letters to denote atomic constants or function symbols, and those beginning with upper case letters to denote variables. We often use specific symbols such as t to denote ground terms, and i, j, k to denote integer indices. We assume an arbitrary but fixed ordering "$\prec$" among variables and ground terms.

A ***type*** is a finite collection of ground terms. In this paper, types represent the space of outcomes of switches or random processes. For example in Palindrome example of Fig. 1, the set of values $\{a,b\}$ is a type. A variable Y referring to the outcome of a switch s is a typed variable; its type, denoted $type(Y)$, is deemed to be the same as the space of outcomes of s. The type of a ground term t can be any of the sets it is an element of.

Definition 1 (*Atomic Constraint*)
An atomic constraint, denoted β, is of the form $\{X = T\}$ or $\{X \neq T\}$, where X is a variable and T is a variable or a ground term of the same type as X.

A set of atomic constraints representing their conjunction is called a ***constraint formula***. Constraint formulas are denoted by symbols γ and ϕ. Note that atomic constraints are closed under negation, while constraint formulas are not. The set of variables in a constraint formula γ are denoted by $Vars(\gamma)$.

Definition 2 (*Constraint Graph*)

The constraint graph for a constraint formula γ is labeled undirected graph, whose nodes are variables and ground terms in γ. Edges are labeled with "$=$" and "$\neq$" such that

- T_1 and T_2 are connected by an edge labeled "$=$" if, and only if, $T_1 = T_2$ is entailed by γ.
- T_1 and T_2 are connected by an edge labeled "$\neq$" if, and only if, $T_1 \neq T_2$ is entailed by γ, and at least one of T_1, T_2 is a variable.

Note that a constraint graph may have edges between two terms even when there is no explicit constraint on the two terms in γ.

Definition 3 (*Ordering*)
Based on an (arbitrary but fixed) total order, "$\prec$", among ground terms, we define an ordering on ground switch instance pairs (s,i) and (s',i') as follows:

- If $i \prec i'$ then $(s,i) \prec (s',i')$ for all ground terms s and s'.
- If $i = i' \wedge s \prec s'$ then $(s,i) \prec (s',i')$.
- If $i = i' \wedge s = s'$ then $(s,i) = (s',i')$.

Canonical representation of constraint formulas. A constraint formula is represented by its constraint graph which in turn can be represented as a set of triples $(source, destination, constraint)$ each of which represents an edge in the constraint graph. Recall that variables and ground terms can be compared using total order $\prec$. Assuming an order between the two symbols "$=$" and "$\neq$", we can define a lexicographic order over each edge triple, and consequently order the elements of the edge set. The sequence of edge triples following the above order is a canonical representation of a constraint formula. Using this representation, we can define a total order, called the *canonical order*, among constraint formulas themselves given by the lexicographic ordering defined over the edge sequences.

Given any constraint formula $\gamma = \{\beta_1, \beta_2, \ldots, \beta_n\}$, its negation $\neg\gamma$ is given by a set of constraint formulas $\neg\gamma = \{\{\neg\beta_1\}, \{\beta_1, \neg\beta_2\}, \ldots, \{\beta_1, \beta_2, \ldots, \beta_{n-1}, \neg\beta_n\}\}$. The above defines negation of a formula to be a set of constraint formulas which are pairwise mutually exclusive and together represent the negation.

The set of solutions of a constraint formula γ is denoted $[\![\gamma]\!]$ and their projection onto a variable $X \in Vars(\gamma)$ is denoted $[\![\gamma]\!]_X$. The constraint formula is unsatisfiable if $[\![\gamma]\!] = \emptyset$, and satisfiable otherwise. Note that substitutions can also be viewed as constraint formulas.

OSDD. We use OSDDs to materialize derivations in a compact manner. OSDDs share a number of features with Binary Decision Diagrams (BDDs) (Bryant 1992) and Multivalued Decision Diagrams (MDDs) (Kam et al. 1998). BDDs are directed acyclic graphs representing propositional boolean formulas, with leaves labeled from $\{0,1\}$ and internal nodes labeled with propositions. In a BDD, each node has two outgoing edges labeled 0 and 1, representing a true and false valuation, respectively, for the node's proposition. An MDD generalizes a BDD where internal nodes are labeled with finite-domain variables, and the outgoing edges are labeled with the possible valuations of that variable. In an OSDD, internal nodes represent switches, and the outgoing edges are labeled with constraints on the outcomes of that node's switch.

Definition 4 (*Ordered Symbolic Derivation Diagram*)
An ordered symbolic derivation diagram over a set of typed variables V is a tree, where leaves are labeled from the set $\{0,1\}$ and internal nodes are labeled by triples of the form (s,k,Y), where s

and k are switch and instance respectively and $Y \in V$. We call Y the output variable of the node. The edges are labeled by constraint formulas over V. We represent OSDDs by textual patterns $(s,k,Y)[\gamma_i : \psi_i]$ where (s,k,Y) is the label of the root, and each sub-OSDD ψ_i is connected to the root by an edge labeled γ_i. OSDDs satisfy the following conditions:

1. **Ordering:** For internal nodes $n = (s,k,Y)$ and $n' = (s',k',Y')$, if n is the parent of n', then $(s,k) \prec (s',k')$. The edges are ordered by using the canonical ordering of the constraint formulas labeling them.
2. **Mutual Exclusion:** The constraints labeling the outgoing edges from an internal node are pairwise mutually exclusive (i.e., $\forall i,j$ such that $i \neq j$, $[\![\gamma_i \wedge \gamma_j]\!] = \emptyset$).
3. **Completeness:** Let $(s,k,Y)[\gamma_i : \psi_i]$ be a sub-OSDD and let σ be any substitution that satisfies all constraints on the path from root to the given sub-OSDD such that $\sigma(Y) \in type(Y)$. Then there is an i such that σ satisfies γ_i.
4. **Urgency:** Let $\mathcal{O}(n)$ be the set of output variables in the path from the root to node n (including n). Then for every constraint formula γ_i labeling an outgoing edge from n, it holds that $Vars(\gamma_i) \subseteq \mathcal{O}(n)$ and for every ancestor n' of n, $Vars(\gamma_i) \not\subseteq \mathcal{O}(n')$.
5. **Explicit Constraints:** If constraint formula γ_i out of a node n entails an implicit atomic constraint β on variables in $\mathcal{O}(n)$, then β occurs explicitly in the path from root to n. A consequence of this condition is that, for every path, the conjunction of constraint formulas labeling edges in the path will be satisfiable.

A tree which satisfies all conditions of an OSDD *except* conditions 4 and 5 is called an *improper OSDD*.

Example 1 (*OSDD properties*)
We illustrate the definiton by using the OSDD shown in Fig. 3(b). The OSDD is represented by the textual pattern $(b,1,X1)[\emptyset : \psi_1]$ where ψ_1 is the sub-tree rooted at the node labeled `msw(b,2,X2)`. This sub-tree, in turn, can be represented by the textual pattern $(b,2,X2)[\{X1 = X2\} : 1, \{X1 \neq X2\} : \psi_2]$ where ψ_2 is the sub-tree rooted at the node `msw(b,3,X3)`, and so on. The internal nodes satisfy the total ordering based on the instances. All outgoing edges from an internal node are pairwise mutually exclusive. For instance, for the outgoing edges of the node labeled `msw(b, 3, X3)`, $X1 = X3$ is mutually exclusive w.r.t. $X1 \neq X3, X2 = X3$ and $X1 \neq X3, X2 \neq X3$. Similarly, $X1 \neq X3, X2 = X3$ is mutually exclusive to $X1 \neq X3, X2 \neq X3$. Consider the sub-OSDD rooted at `msw(b, 2, X2)` any substitution that grounds $X1, X2$ satisfies the (empty) constraints on the path from the root to that subtree. Further any such substitution will satisfy exactly one of the edge constraints $X1 = X2$ or $X1 \neq X2$. It is obvious from the example that urgency is satisfied. Finally consider the constraint formula $X1 \neq X3, X2 = X3$. This entails the implicit constraint $X1 \neq X2$. However, this constraint is explicitly found in the path from the root to that subtree. Therefore the OSDD in Fig. 3(b) is a proper OSDD.

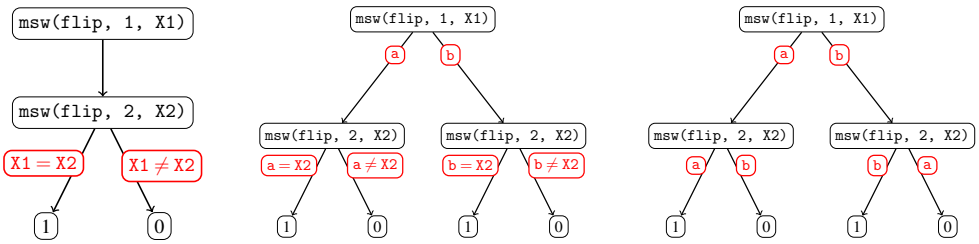
OSDDs can be viewed as representing a set of explanations or derivations where a node of the form (s,k,Y) *binds* Y. This observation leads to the definition of bound and free variables:

Definition 5 (*Bound and Free variables*)
Given an OSDD $\psi = (s,k,Y)[\gamma_i : \psi_i]$ the bound variables of ψ, denoted $BV(\psi)$, are the output variables in ψ. The free variables of ψ, denoted $FV(\psi)$, are those variables which are not bound.

Each OSDD corresponds to an MDD which can be constructed as follows.

Definition 6 (*Grounding*)
Given an OSDD $\psi = (s,k,Y)[\gamma_i : \psi_i]$, the MDD corresponding to it is denoted $\mathcal{G}(\psi)$ and is recursively defined as $\mathcal{G}(\psi) = (s,k,Y)[\alpha_j : \mathcal{G}(\psi_k[\alpha_j/Y])]$ where $\alpha_j \in type(Y)$ and ψ_k is the sub-tree of ψ such that $\gamma_k[\alpha_j/Y]$ is satisfiable.

Example 2 (*Grounding*)
Consider the OSDD in Fig. 4(a). In the first step of the grounding, all values satisfy the (empty) constraint of the outgoing edge. Therefore we get two subtrees which are identical except the substitution that is applied to the variable $X1$ (see Fig. 4(b)). In the next step, the subtrees are ground. Consider the left subtree in Fig. 4(b). The value "a" satisfies the left branch, while the value "b" satisfies the right branch. In a similar fashion the right subtree is ground to yield the ground MDD in Fig. 4(c).



(a) OSDD (b) Partial grounding (c) Ground MDD

Fig. 4. Example of grounding OSDD

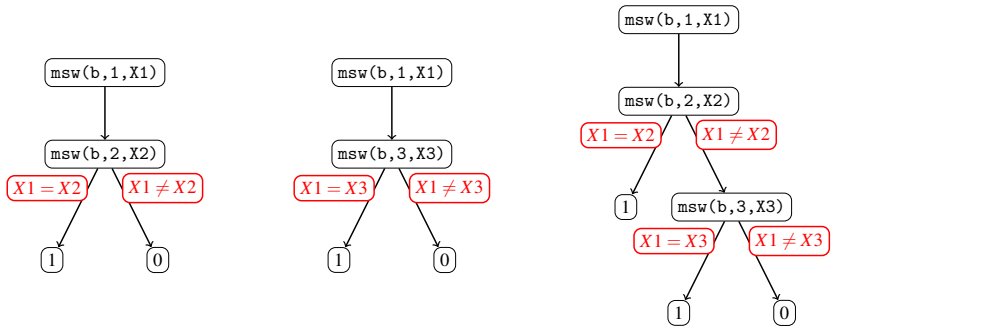
Canonical OSDD representation. Given a total order among variables and terms, the order of nodes in an OSDD is fixed. We can further order the outgoing edges uniquely based on the total order of constraints labeling them. This yields a canonical representation of OSDDs. In the rest of the paper we assume that OSDDs are in canonical form.

We now define common operations over OSDDs which are applied to construct OSDDs from primitives.

Definition 7 (*Conjunction/Disjunction*)
Given OSDDs $\psi = (s,k,Y)[\gamma_i : \psi_i]$ and $\psi' = (s',k',Y')[\gamma'_j : \psi'_j]$, let $\oplus$ stand for either $\wedge$ or $\vee$ operation. Then $\psi \oplus \psi'$ is defined as follows.

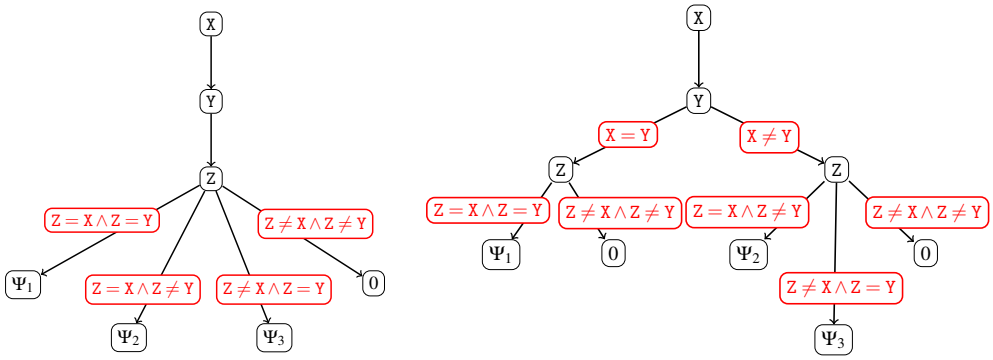
- If $(s,k) \prec (s',k')$, then $\psi \oplus \psi' = (s,k,Y)[\gamma_i : \psi_i \oplus \psi']$
- If $(s',k') \prec (s,k)$ then $\psi \oplus \psi' = (s',k',Y')[\gamma'_j : \psi'_j \oplus \psi]$
- If $(s,k) = (s',k')$, first we apply the substitution $[Y/Y']$ to the second OSDD. Then $\psi \oplus \psi' = (s,k,Y)[\gamma_i \wedge \gamma'_j : \psi_i \oplus \psi'_j]$.

Example 3 (*Conjunction/Disjunction*)
Consider the input OSDDs in Fig. 5(a). Their disjunction is shown in Fig. 5(b). Disjunction of this OSDD with a third OSDD which is isomorphic to the input OSDDs but involving switch instance pairs $(b,2),(b,3)$ results in the OSDD shown in Fig. 3(b).



(a) Input OSDDs (b) Disjunction of input OSDDs

Fig. 5. Disjunction of OSDDs



(a) Improper OSDD (b) Proper OSDD

Fig. 6. Transformation example

Transformation from improper to proper OSDDs. When performing conjunction/disjunction operations over proper OSDDs, the resulting OSDD may be improper. For instance, consider one OSDD with variables X, Z, and another with variables Y, Z. Constraints between X and Z, and those between Y and Z may imply constraints between X and Y that may not be explicitly present in the resulting OSDD, thereby violating the condition of explicit constraints (Def. 4).

Fig. 6(a) shows an improper OSDD that violates the explicit constraints condition. For example, the edge (Z, Ψ_1), leading to the sub-OSDD Ψ_1, has edge constraints which imply $X = Y$ while the edges leading to Ψ_2 and Ψ_3 imply $X \neq Y$.

An improper OSDD can be converted into a proper one by rewriting it using a sequence of steps as follows. First, we identify an implicit constraint and where it may be explicitly added without violating the urgency property. In the example, we identify $X = Y$ as implicit, and attempt to introduce it on the outgoing edges of node labeled Y. This introduction splits the edge from Y to Z into two: one labeled $X = Y$, and another labeled $X \neq Y$, the negation of the identified constraint. The original child rooted at Z is now replicated due to this split. We process each child, eliminating edges and corresponding sub-OSDDs whenever the constraints are unsatisfiable. In the example in Fig. 6(b) we see that the edges (Z, Ψ_2) and (Z, Ψ_3) have been removed from Ψ_{Z_1} since their constraint formula are inconsistent with $X = Y$. We repeat this procedure until no implicit constraints exist in Ψ.

Although OSDDs have been defined as trees, we can turn them into DAGs by combining equivalent subtrees.

Definition 8 (*Equivalence*)
All OSDD leaves which have the same node label are equivalent. Two OSDDs $\psi = (s,k,Y)[\gamma_i : \psi_i]$ and $\psi' = (s,k,Y')[\gamma'_i : \psi'_i]$ are equivalent if $\forall i.\ [\gamma_i : \psi_i] = [\gamma'_i : \psi'_i][Y/Y']$.

It is easy to generalize the OSDD operations defined above to work directly over DAGs.

Constraint application. Given an OSDD, the MDD represented by it is altered by the application of new atomic constraints. The operation which specializes an OSDD to satisfy an atomic constraint is defined as follows.

Definition 9 (*Constraint application*)
Given an OSDD $\psi = (s,k,Y)[\gamma_i : \psi_i]$ and an atomic constraint β, the application of β to ψ results in a new OSDD ψ' as follows:

- Application of β to $0,1$ yields $0,1$ respectively.
- If $Vars(\beta) \subseteq \mathscr{O}(n)$ where n is the root of ψ, then $\psi' = (s,k,Y)[\gamma_i \wedge \beta : \psi_i, \neg\beta : 0]$
- Else $\psi' = (s,k,Y)[\gamma_i : \psi'_i]$ where each ψ'_i results from the application of β to ψ_i.

Properties. OSDDs and the operations defined above have a number of properties necessary for their use in representing explanations for query evaluation in PLPs.

Proposition 1 (*Closure*)
OSDDs are closed under conjunction and disjunction operations.

The following shows that conjunction and disjunction operations over OSDDs lift the meaning of these operations over ground MDDs.

Proposition 2 (*Compatibility with Grounding*)
Let $\psi = (s,k,Y)[\gamma_i : \psi_i]$ and $\psi' = (s',k',Y')[\gamma'_j : \psi'_j]$ be two OSDDs, then

$$\mathscr{G}(\psi \oplus \psi') = \mathscr{G}(\psi) \oplus \mathscr{G}(\psi').$$

3 Construction

Given a definite PLP program and a ground query, query evaluation proceeds by first constructing OSDDs for thw query's answers. The construction is done via constraint-based tabled evaluation over a transformed program. At a high level, each n-ary predicate p/n in the original PLP program is transformed into a $n+2$-ary predicate $p/(n+2)$ with one of the new arguments representing an OSDD at the time of call to p, and the other representing OSDDs for answers to the call.

For simplicity, although the transformed program represents OSDDs as Prolog terms, we reuse the notation from Section 2 to describe the transformation.

Transformation. We use $\overline{T_1}, \overline{T_2}, \ldots$ to represent tuples of arguments. Clauses in a definite program may be of one of two forms:

- Fact $p(\overline{T})$: is transformed to another fact of the form $p(\overline{T}, \mathtt{0}, \mathtt{0})$, denoting that a fact may bind its arguments but does not modify a given OSDD.
- Clause *head* $\leftarrow$ *body*: without loss of generality, we assume that the body is binary: i.e., clauses are of the form $p(\overline{T}) \leftarrow q(\overline{T_1}), r(\overline{T_2})$. Such clauses are transformed into $p(\overline{T}, \mathtt{O_1}, \mathtt{O_3}) \leftarrow q(\overline{T_1}, \mathtt{O_1}, \mathtt{O_2}), r(\overline{T_2}, \mathtt{O_2}, \mathtt{O_3})$

For each user-defined predicate p/n in the input program, we add the following directive for the transformed predicate $p/(n+2)$

```
:- table p(_,...,_, lattice(or/3)).
```

which invokes answer subsumption (Swift and Warren 2010) to group all answers by their first $n+1$ arguments, and combine the $n+2$-nd argument in each group using `or/3`, which implements the disjunction operation over OSDDs.

Constraint-Based Evaluation. An important aspect of OSDD construction is constraint processing. Our transformation assumes that constraints are associated with variables using their *attributes* (Holzbaur 1992). We assume the existence of the following two predicates:

- `inspect(X, C)`, which, given a variable `X`, returns the constraint formula associated with `X`; and
- `store(C)`, which, given a constraint formula `C`, annotates all variables in `C` with their respective atomic constraints.

For tabled evaluation, we assume that each table has a local constraint store (Sarna-Starosta and Ramakrishnan 2007). Such a constraint store can be implemented using the above two predicates.

OSDD Builtins. The construction of the transformed program is completed by defining predicates to handle the two constructs that set PLPs apart:

- `msw(S, K, X, O1, O1)`: Note that `msw`'s in the body of a clause would have been transformed to a 5-ary predicate. This predicate is defined as:

$$\begin{aligned} msw(S,K,Y,\mathtt{O_1},\mathtt{O_2}) \leftarrow & \mathtt{inspect}(Y,\gamma), \\ & (\gamma = \{\} \rightarrow O = (S,K,Y)[\{\}:1] \\ & ;\ O = (S,K,Y)[\gamma:1,\neg\gamma:0] \\ &), \mathtt{and}(\mathtt{O_1},\mathtt{O},\mathtt{O_2}). \end{aligned}$$

  where `and/3` implements conjunction operation over OSDDs.
- Constraint handling: constraints in the input program will be processed using:

$$\begin{aligned} constraint(C,\mathtt{O_1},\mathtt{O_2}) \leftarrow & ((Vars(C) \cap BV(\mathtt{O_1})) \neq \emptyset \\ & \rightarrow \mathtt{O_2} = \mathtt{applyConstraint}(C,\mathtt{O_1}) \\ & ;\ \mathtt{O_2} = \mathtt{O_1} \\ &), \mathtt{store}(C). \end{aligned}$$

  where `applyConstraint` is an implementation of Defn. 9

To compute the OSDD for a ground atom $q(\overline{X})$ in the original program, we evaluate $q(\overline{X},1,O)$ to obtain the required OSDD as O.

4 Inference

The second step in query evaluation computes the probability of each computed answer based on the associated OSDD. Below, we describe exact as well as approximate methods for compute the probabilities.

4.1 Exact Inference

Given an OSDD $\psi = (s,k,Y)[\gamma_i : \psi_i]$, let $Dom(FV(\psi))$ be the Cartesian product of the types of each $X \in FV(\psi)$. We define a function π which maps an OSDD ψ and a substitution $\sigma \in Dom(FV(\psi))$ to $\mathbb{R}$. Its definition for leaves is as follows

$$\pi(1,\sigma) = 1 \text{ and } \pi(0,\sigma) = 0.$$

Next for an OSDD ψ and an arbitrary substitution σ', define $\Pi(\psi,\sigma')$ to be

$$\Pi(\psi,\sigma') = \sum_{\sigma:\sigma \nparallel \sigma'} \pi(\psi,\sigma),$$

where constraint formulas σ and σ' are said to be compatible if their conjunction is satisfiable (denoted $\sigma \nparallel \sigma'$). For internal nodes we define $\pi(\psi,\sigma)$ as

$$\pi(\psi,\sigma) = \sum_i \sum_{y \in [\![\gamma_i\sigma]\!]_Y} P(Y=y)\Pi(\psi_i,\sigma[y/Y]).$$

where $P(Y=y)$ is the probability that k-th instance of s has outcome y.

Given an answer whose OSDD is ψ, we return the answer probability as $\Pi(\psi,\emptyset)$.

Proposition 3 (*Complexity*)
The time complexity of probability computation of OSDD ψ is $O(D \cdot N \cdot exp(V))$ where D is the maximum cardinality of all types, N is the number of nodes in ψ, and V is the size of the largest set of free variables among all internal nodes of ψ.

Under certain conditions we can avoid the exponential complexity of the naive probabilistic inference algorithm. By exploiting the regular structure of the solution space to a constraint formula we avoid the explicit summation $\sum_{y \in [\![\gamma_i\sigma]\!]_Y}$. In this case we say that γ_i is *measurable*. We formally define measurability and a necessary and sufficient condition for checking if a constraint is measurable.

Definition 10 (*Measurability*)
A satisfiable constraint formula γ is said to be measurable w.r.t $X \in Vars(\gamma)$ if for all ground substitutions σ on $Vars(\gamma) \setminus \{X\}$ which satisfy γ, $|[\![\gamma\sigma]\!]_X|$ is equal to a unique value m_X called the measure of X in γ.

Definition 11 (*Saturation*)
A constraint formula γ is said to be saturated if its constraint graph satisfies the following condition: For every $X \in Vars(\gamma)$, let $\mathscr{Z}$ be the set of nodes connected to X with a " $\neq$ " edge. Then there exists an edge (" $\neq$ " or " $=$ ") between each pair of nodes in $\mathscr{Z}$ (except when both nodes in the pair represent constants).

Proposition 4 (*Condition for Measurability*)
A satisfiable constraint formula is measurable w.r.t all of its variables if and only if it saturated.

Definition 12 (*Measurability of OSDDs*)
An OSDD is said to be measurable, if for each internal node n and outgoing edge labeled γ_i, the constraint formula obtained by taking the conjunction of γ_i with the constraint formula on the path from root to n is measurable w.r.t the output variable in node n.

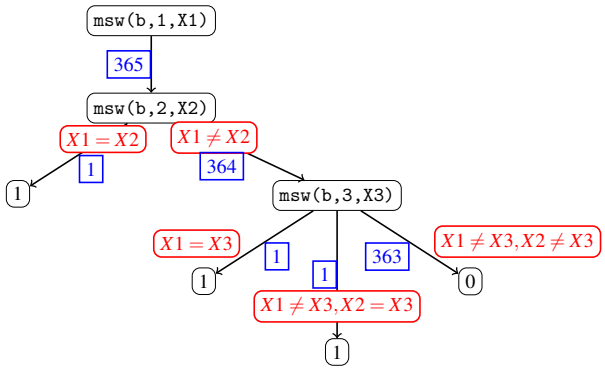


Fig. 7. Measures for birthday OSDD

Example 4 (*Measurability*)
As an example, consider the OSDD for the birthday problem shown in Fig. 3(b). The OSDD is measurable according to Def. 12. Consider for instance the node labeled $msw(b,3,X3)$ and the outgoing edge labeled $X1 \neq X3, X2 = X3$. The constraint formula obtained by taking the conjunction of the root to node path constraints with the edge constraints is $X1 \neq X2, X1 \neq X3, X2 = X3$. This formula is saturated and is therefore measurable w.r.t $X3$. In a similar fashion we can verify that the OSDD is measurable. The measures of the output variables for each of the edges is shown in Fig. 7 (the measures are in rectangular boxes).

When an OSDD is measurable and all distributions are uniform, the probability computation gets specialized as follows:

$$\pi(\psi,\sigma) = \sum_i m_i P(Y = \hat{y}_i)\Pi(\psi_i, \sigma[\hat{y}_i/Y])$$

where $\hat{y}_i$ is an arbitrary value in $[\![\gamma_i\sigma]\!]_Y$ and $P(Y = \hat{y}_i) = \frac{1}{|type(Y)|}$.

Proposition 5 (*Complexity for Measurable OSDDs*)
If an OSDD ψ is measurable and all switches have uniform distribution, the complexity of computing probability of ψ is $O(D \cdot N)$ where D is the maximum cardinality of all types and N is the number of nodes in ψ.

4.2 Likelihood-Weighted Sampling

Likelihood weighting is a popular sampling-based inference technique in Bayesian Networks. The technique can be described as follows: sample all variables except evidence variables in the topological order. The values of evidence variables are fixed and each one of them contributes a *likelihood weight*, which is the probability of its value given the values of its parents. The likelihood weight of the entire sample is the product of the likelihood weights of all evidence variables. This technique has been shown to produce sample estimates with lower variance than independent sampling (Fung and Chang 1990; Shachter and Peot 1990).

Likelihood weighted sampling can be generalized to PLPs as follows: Given an OSDD $\psi = (s,k,Y)[\gamma_i : \psi_i]$ for evidence, generate a sample as follows:

- Construct $type'(Y) = type(Y) \setminus \cup_j [\![\gamma_j]\!]_Y$ where $\psi_j = 0$.

- If $type'(Y) = type(Y)$ sample y from the distribution of Y leaving likelihood weight of the sample unchanged.
- Otherwise, sample y uniformly from $type'(Y)$ and multiply the likelihood weight of the sample by $P(Y = y)$. Let $y \in [\![\gamma_i]\!]_Y$ for some i. Then continue construction of the sample by recursively sampling from the OSDD $\psi_i[y/Y]$.

For PLPs encoding Bayesian Networks and Markov Networks, the simple nature of evidence allows us to generate only consistent samples. However, for general queries in PLP, it is possible to reach a node whose edge constraints to non-0 children are unsatisfiable. In that case, we reject the current sample and restart. Thus, we generalize traditional LW sampling.

To compute conditional probabilities, samples generated for evidence are extended by evaluating queries. The conditional probability of the query given evidence is computed as the sum of the likelihood weights of the samples satisfying the query and evidence divided by the sum of the likelihood weights of the samples which satisfy evidence. To compute unconditional probability of a query, we simply compute the average likelihood weight of the samples satisfying the query.

Example 5 (*Likelihood weighting*)
Consider the OSDD shown in Fig. 3(a). To generate a likelihood weighted sample we start from the root and sample the random variables. The first three nodes do not have any constraints on their outgoing edges, therefore we can sample those random variables from their distributions. Assume that we get the sequence "aba". The likelihood weight of the sample remains 1 at this stage. When sampling the random variables at the next three nodes, $type'(Y)$ gets restricted to a single value. Since the distributions are all uniform the likelihood of the entire sample becomes 0.5^3. All samples would have the same likelihood weight and therefore the probability of "`evidence(6)`" is 0.125.

5 Experimental Evaluation

In this section we present experimental results evaluating the performance of exact inference and likelihood weighted sampling, and comparing them with the inference using other PLP systems. Our prototype implementation was written in XSB Prolog and supported by modules written in C. The following problems were used in the experiments.

- **Palindrome**, which is shown in Fig. 1, with evidence limited to strings of length $N = 20$, and query checking $K = 4$ "`a`"s. While the compuation of evidence probability is easy, the computation of the probability for conjunction of query and evidence is not. This is because the query searches for all possible combinations of 'K' positions. Therefore, for large problem sizes, likelihood weighting must be used to produce approximate answers.
- **Birthday**, shown in Fig. 2 with population size of 6, i.e. query `same_birthday(6)`. While this query can be evaluated by exact inference due to measurability, we also used it to test the performance of the likelihood weighted sampler.

Exact inference. Table 1 shows the time required to construct the OSDDs and perform inference for the two example problems[1]. The leftmost column gives the problem size, viz., population size (Birthday) or length of string (Palindrome). While the Birthday problem has a single OSDD, the

[1] We used a core i5 machine with 16 GB memory running macOS 10.13.4

	Birthday		Palindrome		
Size	OSDD	M. prob.	evid. OSDD	qe OSDD	M. prob
10	0.215	0.014	0.011	0.719	0.008
12	0.795	0.024	0.013	1.877	0.008
14	5.49	0.035	0.014	4.94	0.033
16	52.83	0.056	0.017	16.03	0.076

Table 1. *Time for OSDD generation and probability computation (seconds)*

Palindrome problem requires two OSDDs: the one for evidence ("evid. OSDD") and one for conjunction of query and evidence ("qe OSDD"). For Palindrome, while "evid. OSDD" scales well with problem size, "qe OSDD" grows quickly. However, all of these OSDDs satisfy the measurability property. Thus it is feasible to perform exact inference as long as the OSDD can be generated. The columns with title "M. prob" show the time required to compute the probability from the OSDDs by exploiting measurability. These results show that exact inference is able to scale to reasonable problem sizes by exploiting measurability.

We evaluated these problems using exact inference in ProbLog and PITA[2]. On the Birthday problem, neither system could complete inference for population size greater than 2. On the Palindrome problem, ProbLog could complete inference for $n \leq 10$ within 15 minutes; and PITA for $n \leq 18$ in the same time.

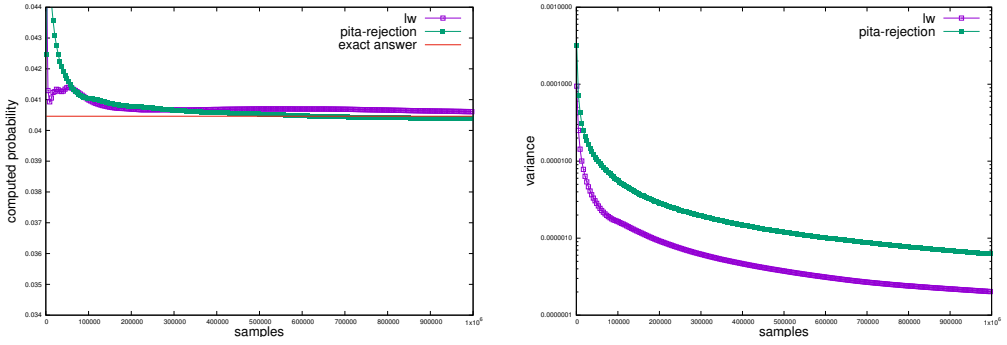
Sampling-Based Inference. We evaluated the performance of LW sampler with rejection sampling implemented in our prototype, and compared with the sampling-based inference of ProbLog and PITA. The time taken to generate samples are shown in Table 2. Columns labeled "OSDD gen." and "LW" show the times taken to construct the OSDD and to generate a single sample in our likelihood weighted sampler. Thanks to constraint propagation in OSDDs, all samples we generate in the two examples are consistent with evidence. The time taken for ProbLog's rejection sampler to generate a single sample, whether or not consistent with evidence, is shown in "ProbLog" column. ProbLog offers an option to propagate evidence, but that was not effective in the two examples, failing to generate consistent samples within 5 minutes. The average time to generate a consistent sample with PITA's rejection sampler (Riguzzi 2011) is shown in "PITA-rej." column. PITA also offers a Metropolis-Hastings sampler, which however shows slower sampling times, and is not included in our experiments.

In the Birthday example, we compute an unconditional probability. Nevertheless, we can use likelihood weighting to compute the answer probability as described earlier. Figs. 8(a) and 8(b). show the computed probability and the variance of the estimates. Since the query is unconditional, no sample is rejected, and hence likelihood weighting does not perform much better than rejection sampling of PITA. Interestingly, sampling using the OSDD structure was significantly faster (up to $2\times$) than using the program directly. This is because the program's non-deterministic evaluation has been replaced by a deterministic traversal through the OSDD.

[2] ProbLog version: 2.1.0.19 and PITA available with SWI-Prolog 7.6.4

Problem	OSDD gen.	LW	ProbLog	PITA-rej.
palindrome	0.019	0.00017	188	0.41
birthday	0.025	1.7e-5	19	0.004

Table 2. *Time for OSDD computation and per sample (seconds)*



(a) Birthday probability (b) Birthday variance

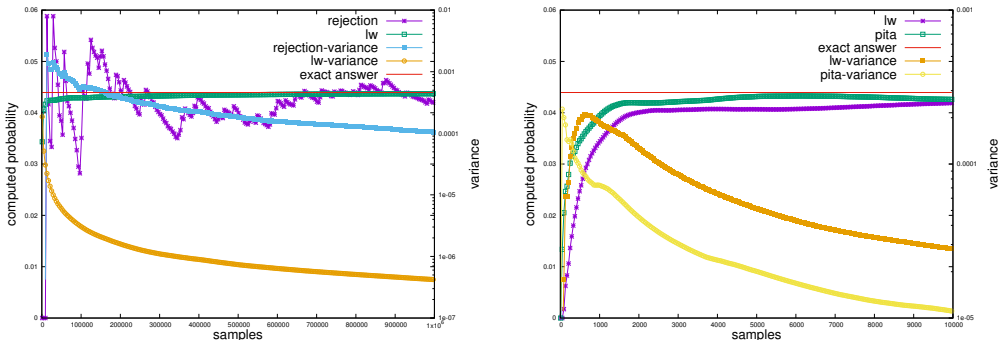
Fig. 8. Birthday sampling results

We evaluate the performance of the LW sampler on the Palindrome example in two ways. Firstly, we compare the computed probability and variance of estimates for LW and rejection sampling implemented in our prototype (Fig. 9(a)). Here we clearly see that LW sampler converges to the correct answer with lower variance, while the rejection sampler fails to converge even after a million samples. Secondly, we compare the computed probability and variance of estimates for LW and PITA's rejection sampler (Fig. 9(b)). Since the sample generation for PITA is significantly slower than LW sampling (see Table 2), Fig. 9(b) is shown for far fewer samples (10K vs 1M) compared to that in Fig. 9(a). From the figure, observe that for the same number of samples, PITA's rejection sampler exhibits smaller variance. This surprising result is an artifact of how samples are counted in PITA: samples that are rejected are not counted (in contrast to traditional rejection samplers). However, note that in a given amount of time, we can draw three orders of magnitude more *consistent* samples with LW than with PITA. Consequently, when run for a fixed amount of time, LW exhibits lower variance than PITA.

6 Related Work

Symbolic inference based on OSDDs was first proposed by Nampally and Ramakrishnan (2015). The present work expands on it two significant ways: Firstly, the construction of OSDDs is driven by tabled evaluation of transformed programs instead of by abstract resolution. Secondly, we give an exact inference procedure for probability computation using OSDDs which generalizes the exact inference procedure with ground explanation graphs.

Probabilistic Constraint Logic Programming (Michels et al. 2013) extends PLP with constraint logic programming (CLP). It allows the specification of models with imprecise probabilities. Whereas a world in PLP denotes a specific assignment of values to random variables, a world in



(a) Palindrome Rejection vs LW (b) Palindrome PITA vs LW

Fig. 9. Palindrome sampling results

PCLP can define constraints on random variables, rather than specific values. Lower and upper bounds are given on the probability of a query by summing the probabilities of worlds where query follows and worlds where query is possibly true respectively. While the way in which "proof constraints" of a PCLP query are obtained is similar to the way in which symbolic derivations are obtained (i.e., through constraint based evaluation), the inference techniques employed are completely different with PCLP employing satisfiability modulo theory (SMT) solvers.

cProbLog extends ProbLog with first-order constraints (Fierens et al. 2012). This gives the ability to express complex evidence in a succinct form. The semantics and inference are based on ProbLog. In contrast, our work makes the underlying constraints in a query explicit and uses the OSDDs to drive inference.

CLP($\mathscr{BN}$) (Costa et al. 2002) extends logic programming with constraints which encode conditional probability tables. A CLP($\mathscr{BN}$) program defines a joint distribution on the ground skolem terms. Queries are answered by performing inference over a corresponding BN.

There has been a significant interest in the area of lifted inference as exemplified by the works of Poole (2003), Braz et al. (2005) and Milch et al. (2008). The main idea of lifted inference is to treat indistinguishable *instances* of random variables as one unit and perform inference at the population level. Lifted inference in the context of PLP has been performed by converting the problem to parfactor representation (Bellodi et al. 2014) or weighted first-order model counting (den Broeck et al. 2011). Lifted explanation graphs (Nampally and Ramakrishnan 2016) are a generalization of ground explanation graphs, which treat instances of random processes in a symbolic fashion. In contrast, exact inference using OSDDs treats *values* of random variables symbolically, thereby computing probabilities without grounding the random variables. Consequently, the method in this paper can be used when instance-based lifting is inapplicable. Its relationship to more recent liftable classes (Kazemi et al. 2016) remains to be explored.

The use of sampling methods for inference in PLPs has been widespread. The evidence has generally been handled by heuristics to reduce the number of rejected samples (Cussens 2000; Moldovan et al. 2013). More recently, Nitti et al. (2016) present an algorithm that generalizes the applicability of LW samples by recognizing when valuation of a random variable will lead to query failure. Our technique propagates constraints imposed by evidence. With a rich constraint language and a propagation algorithm of sufficient power, the sampler can generate consistent samples without any rejections.

Adaptive sequential rejection sampling (Mansinghka et al. 2009) is an algorithm that adapts its proposal distributions to avoid generating samples which are likely to be rejected. However, it

requires a decomposition of the target distribution, which may not be available in PLPs. Further, in our work the distribution from which samples are generated is not adapted. It is an interesting direction of research to combine adaptivity with the proposed sampling algorithm.

7 Conclusion

In this work we introduced OSDDs as an alternative data structure for PLP. OSDDs enable efficient inference over programs whose random variables range over large finite domains. We also showed the effectiveness of using OSDDs for likelihood weighted sampling.

OSDDs may provide asymptotic improvements for inference over many classes of first-order probabilistic graphical models. An example of such models is the Logical hidden Markov model (LOHMM) which lifts the representational structure of hidden Markov models (HMMs) to a first-order domain (Kersting et al. 2006). LOHMMs have proved to be useful for applications in computational biology and sequential behavior modeling. LOHMMs encode first-order relations using *abstract transitions* of the form $p : \mathtt{H} \overset{\mathtt{O}}{\leftarrow} \mathtt{B}$ where $p \in [0,1]$. Any of $\mathtt{H}, \mathtt{B}, \mathtt{O}$ may be partially ground, and there may be logical variables which are shared between any of the atoms in an abstract transition. Abstract explanations that are obtained by inference over such models that avoids grounding of variables whenever possible can be naturally captured by OSDDs.

Acknowledgements

This work was support in part by NSF grant IIS-1447549. We are grateful for the reviewers for their detailed and insightful comments.

References

BELLODI, E., LAMMA, E., RIGUZZI, F., COSTA, V. S., AND ZESE, R. 2014. Lifted variable elimination for probabilistic logic programming. *Theory and Practice of Logic Programming 14,* 4-5, 681–695.

BRAZ, R. D. S., AMIR, E., AND ROTH, D. 2005. Lifted first-order probabilistic inference. In *19th International Joint Conference on Artificial intelligence*. 1319–1325.

BRYANT, R. E. 1992. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys 24,* 3, 293–318.

COSTA, V. S., PAGE, D., QAZI, M., AND CUSSENS, J. 2002. CLP(BN): Constraint logic programming for probabilistic knowledge. In *19th Conference on Uncertainty in Artificial Intelligence*. 517–524.

CUSSENS, J. 2000. Stochastic logic programs: Sampling, inference and applications. In *16th Conference on Uncertainty in Artificial Intelligence*. Morgan Kaufmann, 115–122.

DEN BROECK, G. V., TAGHIPOUR, N., MEERT, W., DAVIS, J., AND RAEDT, L. D. 2011. Lifted probabilistic inference by first-order knowledge compilation. In *20th International Joint Conference on Artificial Intelligence*. 2178–2185.

FIERENS, D., DEN BROECK, G. V., BRUYNOOGHE, M., AND RAEDT, L. D. 2012. Constraints for probabilistic logic programming. In *NIPS Probabilistic Programming Workshop*. 1–4.

FUNG, R. M. AND CHANG, K.-C. 1990. Weighing and integrating evidence for stochastic simulation in bayesian networks. In *5th Conference on Uncertainty in Artificial Intelligence*. 209–220.

GEMAN, S. AND GEMAN, D. 1984. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence 6,* 6, 721–741.

HASTINGS, W. K. 1970. Monte carlo sampling methods using markov chains and their applications. *Biometrika 57,* 1, 97–109.

HOLZBAUR, C. 1992. Metastructures versus attributed variables in the context of extensible unification. In *4th International Symposium on Programming Language Implementation and Logic Programming*. 260–268.

KAM, T., VILLA, T., BRAYTON, R., AND SANGIOVANNI-VINCENTELLI, A. 1998. Multivalued decision diagrams: Theory and applications. *Multiple-Valued Logic 4*, 9–62.

KAZEMI, S. M., KIMMIG, A., DEN BROECK, G. V., AND POOLE, D. 2016. New liftable classes for first-order probabilistic inference. In *30th International Conference on Neural Information Processing Systems*. 3125–3133.

KERSTING, K., RAEDT, L. D., AND RAIKO, T. 2006. Logical hidden Markov models. *Journal of Artificial Intelligence Research 25*, 2006.

MANSINGHKA, V., ROY, D., JONAS, E., AND TENENBAUM, J. 2009. Exact and approximate sampling by systematic stochastic search. In *12th International Conference on Artificial Intelligence and Statistics*. 400–407.

MICHELS, S., HOMMERSOM, A., LUCAS, P. J., VELIKOVA, M., AND KOOPMAN, P. W. 2013. Inference for a new probabilistic constraint logic. In *23rd International Joint Conference on Artificial Intelligence*. 2540–2546.

MILCH, B., ZETTLEMOYER, L. S., KERSTING, K., HAIMES, M., AND KAELBLING, L. P. 2008. Lifted probabilistic inference with counting formulas. In *23rd AAAI Conference on Artificial Intelligence*. 1062–1068.

MOLDOVAN, B., THON, I., DAVIS, J., AND RAEDT, L. D. 2013. MCMC estimation of conditional probabilities in probabilistic programming languages. In *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*. 436–448.

NAMPALLY, A. AND RAMAKRISHNAN, C. R. 2015. Constraint-based inference in probabilistic logic programs. In *2nd International Workshop on Probabilistic Logic Programming*. 46–56.

NAMPALLY, A. AND RAMAKRISHNAN, C. R. 2016. Inference in probabilistic logic programs using lifted explanations. In *Technical Communications of 32nd International Conference on Logic Programming*. 15:1–15:15.

NITTI, D., LAET, T. D., AND RAEDT, L. D. 2016. Probabilistic logic programming for hybrid relational domains. *Machine Learning 103,* 3, 407–449.

POOLE, D. 2003. First-order probabilistic inference. In *18th International Joint Conference on Artificial Intelligence*. 985–991.

RAEDT, L. D., KIMMIG, A., AND TOIVONEN, H. 2007. ProbLog: a probabilistic Prolog and its application in link discovery. In *20th International Joint Conference on Artifical Intelligence*. 2462–2467.

RIGUZZI, F. 2011. MCINTYRE: A Monte Carlo algorithm for probabilistic logic programming. In *26th Italian Conference on Computational Logic (CILC2011)*. 25–39.

RIGUZZI, F. AND SWIFT, T. 2011. The PITA system: Tabling and answer subsumption for reasoning under uncertainty. *Theory and Practice of Logic Programming 11,* 4-5, 433–449.

SARNA-STAROSTA, B. AND RAMAKRISHNAN, C. R. 2007. Compiling constraint handling rules for efficient tabled evaluation. In *Practical Aspects of Declarative Languages (PADL)*. 170–184.

SATO, T. AND KAMEYA, Y. 1997. PRISM: a language for symbolic-statistical modeling. In *15th International Joint Conference on Artificial Intelligence*. 1330–1339.

SATO, T. AND KAMEYA, Y. 2001. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research 15*, 391–454.

SHACHTER, R. D. AND PEOT, M. A. 1990. Simulation approaches to general probabilistic inference on belief networks. In *5th Conference on Uncertainty in Artificial Intelligence*. 221–234.

SWIFT, T. AND WARREN, D. S. 2010. Tabling with answer subsumption: Implementation, applications and performance. In *Logics in Artificial Intelligence: 12th European Conference (JELIA)*. 300–312.

TAMAKI, H. AND SATO, T. 1986. OLD resolution with tabulation. In *3rd International Conference on Logic Programming*. Springer, 84–98.