Probabilistic Theorem Proving: A Unifying Approach for Inference in Probabilistic Programming

Vibhav Gogate
The University of Texas at Dallas
vgogate@hlt.utdallas.edu

Pedro Domingos
University of Washington
pedrod@cs.washington.edu

1 Introduction

Inference is the key bottleneck in probabilistic programming. Often, the main advantages of probabilistic programming – simplicity, modularity, ease-of-use, etc. – are dwarfed by the complexity and intractability of inference. In fact, one of the main reasons for the scarcity/absence of large applications and real-world systems that are based in large part on probabilistic programming languages (PPLs) (c.f. [6, 8]) is the lack of fast, scalable and accurate inference engines for them. Therefore, in this paper, we consider *probabilistic theorem proving* (PTP) [5], a recently proposed scalable, general-purpose algorithm for inference in probabilistic logic, and extend it to yield a general-purpose inference algorithm for PPLs.

PTP solves the following problem: compute the probability of a formula given probabilities or weights associated with a set of formulas in first-order logic (or Markov logic [4]). It includes many important problems as special cases, including propositional satisfiability, inference in graphical models and theorem proving (see Fig.1). In our previous work, we showed that the problem addressed by PTP can be reduced to lifted weighted model counting (LWMC). Consequently, algorithms that solve LWMC can be used to solve all of these problems.We proposed both an exact, DPLL/Relsat [1] based search algorithm and an approximate, importance sampling based algorithm for LWMC and showed that they are much superior in terms of accuracy and scalability than state-of-the-art approaches.

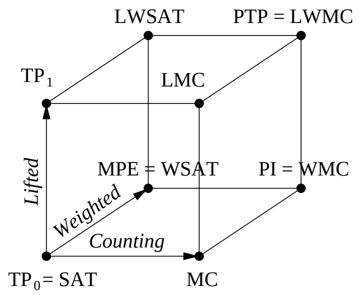

Figure 1: Inference problems addressed by PTP. TP_0 and TP_1 is propositional and first-order theorem proving respectively, PI is probabilistic inference, MPE is computing the most probable explanation, SAT is satisfiability, MC is model counting, W is weighted and L is lifted. $A = B$ means A can be reduced to B.

Our new algorithms based on PTP/LWMC have several desirable properties. First, our LWMC algorithms are *lifted*, namely just like theorem proving they can infer over a group of objects in one shot, often yielding exponential performance gains over propositional algorithms. Since theorem proving serves as a general-purpose inference engine for deterministic programming languages such as Prolog, we envision that PTP/LWMC will serve as a general-purpose inference engine for probabilistic languages. Second, our LWMC algorithms take advantage of context-specific independence (CSI) [2] and determinism [3] by leveraging SAT technology that has matured considerably over the last few decades. Both CSI and determinism are quite common in real-world probabilistic programs. Third, and more importantly there exists an open source software implementation of LWMC, Alchemy 2.0 (available at `http://code.google.com/p/alchemy-2/`) which can help speed up the development time considerably.

2 Our Approach

We propose a two step solution to the inference problem in PPLs. In the first step, we convert the probabilistic program and the query to a (recursive) probabilistic knowledge base (PKB) and in the second step we use an extension of probabilistic theorem proving (PTP) [5] to compute the probability of the query. We need to extend PTP because PTP, in its current form, is not suitable for performing

```
(define Asthma (lambda(x) (flip 0.1)))
(define Smokes (lambda(x)
                    (if (Asthma(x))
                      (flip 0.01) (flip 0.5))))
(define Friends (lambda(x,y)
                      (if (and (Asthma x) (Smokes y))
                          (flip 0.01) (flip 0.5))))
```

(a) A Church Program

$\neg\texttt{Asthma}(x), 0.9$; $\texttt{Asthma}(x), 0.1$;
$\neg\texttt{Asthma}(x) \vee \neg\texttt{Smokes}(x), 0.01$; $\neg\texttt{Asthma}(x) \vee \texttt{Smokes}(x), 0.99$;
$\texttt{Asthma}(x) \vee \neg\texttt{Smokes}(x), 0.5$; $\texttt{Asthma}(x) \vee \texttt{Smokes}(x), 0.5$;
$\neg\texttt{Asthma}(x) \vee \neg\texttt{Smokes}(y) \vee \neg\texttt{Friends}(x,y), 0.01$;
$\neg\texttt{Asthma}(x) \vee \neg\texttt{Smokes}(y) \vee \texttt{Friends}(x,y), 0.99$;
$\neg((\neg\texttt{Asthma}(x) \vee \neg\texttt{Smokes}(y)) \wedge \texttt{Friends}(x,y)), 0.5$
$\neg((\neg\texttt{Asthma}(x) \vee \neg\texttt{Smokes}(y)) \wedge \neg\texttt{Friends}(x,y)), 0.5$

(b) An Equivalent PKB

Figure 2: (a) A Church program that defines a social network about friendship between Asthamtics and Smokers. The first statement says that people generally don't have Asthma. The second statement says that Asthamtics generally do not smoke and the third statement says that people who have Asthma generally are not friends with people who smoke. (b) A PKB that is equivalent to the church program given in (a).

inference in PKBs derived from probabilistic programs. The reason is that probabilistic programs are often self-recursive and have infinite loops. As a result, PKBs derived from them have logical variables with infinite domains, and on such PKBs, both exact and approximate versions of PTP may not halt. We therefore extend PTP to handle such cases.

A PKB [5] is a set of weighted first-order logic formulas. A weighted formula is a pair (F_i, ϕ_i) where F_i is a formula in first-order logic and $\phi_i \geq 0$. *Hard* formulas have $\phi_i = 0$ and *soft* formulas have $\phi_i > 0$. Formally, if $\mathbf{x}$ is a world and $\Phi_i(\mathbf{x})$ is the potential corresponding to formula F_i, then $\Phi_i(\mathbf{x}) = 1$ if F_i is true, and $\Phi_i(\mathbf{x}) = \phi_i$ if the formula is false. We interpret a universally quantified formula as a set of features, one for each grounding of the formula, as in Markov logic [4]. A PKB $\{(F_i, \phi_i)\}$ thus represents the following joint probability distribution $P(\mathbf{x}) = \frac{1}{Z}\prod_i \phi_i^{n_i(\mathbf{x})}$, where $\mathbf{x}$ is a truth-assignment to all groundings of all atoms, $n_i(\mathbf{x})$ is the number of false groundings of F_i in $\mathbf{x}$, and Z is the *partition function*.

2.1 Converting a Probabilistic Program to a PKB

Fig. 2 illustrates the conversion. Fig. 2(a) shows a Church program [6] and Fig. 2(b) shows an equivalent PKB. For each definition in the program, we introduce a first-order atom in the PKB. The terms of the first-order atoms are the inputs to the corresponding definition (or its λ-expression). For each constraint in each definition, we introduce a soft formula in the PKB. For example, for the statement "if (Asthma(x)) (flip 0.01)" in the definition of Smokes, we introduce two soft formulas: $\neg\texttt{Asthma}(x) \vee \neg\texttt{Smokes}(x), 0.01$ and $\neg\texttt{Asthma}(x) \vee \texttt{Smokes}(x), 0.99$.

To handle recursion in probabilistic programs, we introduce some syntactic sugar. To differentiate it from a regular PKB, we will call it *recursive PKB*. We illustrate the key steps in building a recursive PKB from a probabilistic program using the program given in Fig. 3(a). To model the infinite self-recursion, we introduce an atom `SillyGameTmp`, that has three logical variables x, y and t as its terms. As before, the domains of x and y are the objects in the real-world while the domain of t is infinite. Fig. 3(b) shows an equivalent recursive PKB to the church program given in Fig. 3(a). The recursive PKB has four soft formulas and four hard formulas. The soft formulas model the probability distribution associated with the flips. The first hard formula expresses the constraint that if `Flip1` is True then `SillyGameTmp` has to be True. The second hard formula models that constraint that if `Flip1` is False and `Flip2` is True then `SillyGameTmp` has to be True. The third hard formula represents the constraint that if both the flips are False then the value of `SillyGameTmp` at level t is the same as the value returned by `SillyGameTmp` at level $t+1$ (The variable t thus keeps track of the recursion level). The last formula expresses the constraint that the marginal distribution associated with `SillyGame`(x,y) is the same as marginal distribution associated with `SillyGameTmp`(x,y,0).

```
(define SillyGame (lambda(x,y)
  (if (flip 0.3)
    True
    (if (flip 0.7)
      False
      (SillyGame(x,y))))))
```

(a) A Church program

$\neg\texttt{Flip1}(x,y,t), 0.3$; $\texttt{Flip1}(x,y,t), 0.7$
$\neg\texttt{Flip2}(x,y,t), 0.7$; $\texttt{Flip2}(x,y,t), 0.3$
$\neg(\texttt{Flip1}(x,y,t) \Rightarrow \texttt{SillyGameTmp}(x,y,t)), 0$
$\neg(\neg\texttt{Flip1}(x,y,t) \wedge \texttt{Flip2}(x,y,t) \Rightarrow \neg\texttt{SillyGameTmp}(x,y,t)), 0$
$\neg(\neg\texttt{Flip1}(x,y,t) \wedge \neg\texttt{Flip2}(x,y,t) \Rightarrow (\texttt{SillyGameTmp}(x,y,t{+}1) \Leftrightarrow \texttt{SillyGameTmp}(x,y,t))), 0$
$\neg(\texttt{SillyGame}(x,y) \Leftrightarrow \texttt{SillyGameTmp}(x,y,0)), 0$

(b) Equivalent Recursive PKB

Figure 3: (a) A toy self-recursive Church program (b) Equivalent recursive PKB to the program given in (a).

2.2 Answering Queries over the recursive PKB using PTP

PTP takes as input a PKB K and a query formula Q and outputs $P(Q|K)$. Recall that the partition function of a PKB K is given by $Z(K) = \sum_{\mathbf{x}} \prod_i \phi_i^{n_i(\mathbf{x})}$. The conditional probability $P(Q|K)$ is simply a ratio of two partition functions: $P(Q|K) = Z(K \cup \{Q, 0\})/Z(K)$, where $Z(K \cup \{Q, 0\})$ is the partition function of K with Q added as a hard formula.

The main idea in PTP is to compute the partition function of a PKB by performing lifted weighted model counting. The model count of a formula in CNF is the number of worlds that satisfy it. In weighted model counting (WMC) [3, 9], each literal has a weight, and a CNFs count is the sum over satisfying worlds of the product of the weights of the literals that are true in that world. PTP uses a lifted, weighted extension of the Relsat model counter [1], which is in turn an extension of the DPLL satisfiability solver. PTP first converts the PKB into a CNF L and set of literal weights W. To achieve this, we replace each soft formula F_i is replaced by a hard formula $F_i \Leftrightarrow A_i$, where A_i is a new atom with the same arguments as F_i and the weight of A_i is ϕ_i.

Algorithm 1 shows pseudo-code for LWMC, the core routine in PTP. $M(L)$ is the set of atoms appearing in L, and $n_A(S)$ is the number of groundings of A consistent with the substitution constraints in S. A lifted decomposition is a partition of a first-order CNF into a set of CNFs with no ground atoms in common. A lifted split of an atom A for CNF L is a partition of the possible truth assignments to groundings of A such that, in each part, (1) all truth assignments have the same number of true atoms and (2) the CNFs obtained by applying these truth assignments to L are identical. For the ith part, n_i is its size, t_i/f_i is the number of True/False atoms in it, σ_i is some truth assignment in it, $L|\sigma_i$ is the result of simplifying L with σ_i , and S_i is S augmented with the substitution constraints added in the process. In a nutshell, LWMC works by finding lifted splits that lead to lifted decompositions, progressively simplifying the CNF until the base case is reached.

A key sub-task in LWMC is identifying lifted splits and lifted decomposition in time that is polynomial (preferably linear or constant) in the number of atoms in L. In [5, 7], we proposed a method for each. Both methods are linear in the size of L in absence of substitution constraints and can yield exponential speedups over propositional WMC algorithms such as Cachet [9] and ACE [3].

Algorithm 1 **LWMC**(CNF C, substs. S, weights W)

```
// Lifted base case
if all clauses in $C$ are satisfied then
    return $\prod_{A \in \mathbf{A}(C)} (W_A + W_{\neg A})^{n_A(S)}$
if $C$ has an empty unsatisfied clause then return 0
// Lifted decomposition step
if there exists a lifted decomposition $\{C_{1,1}, \ldots, C_{1,m_1},$
    $\ldots, C_{k,1}, \ldots, C_{k,m_k}\}$ of $C$ under $S$ then
    return $\prod_{i=1}^{k} [\text{LWMC}(C_{i,1}, S, W)]^{m_i}$
// Lifted splitting step
Choose an atom $A$
Let $\{\Sigma_{A,S}^{(1)}, \ldots, \Sigma_{A,S}^{(l)}\}$ be a lifted split of $A$ for $C$ under $S$
return $\sum_{i=1}^{l} n_i W_A^{t_i} W_{\neg A}^{f_i} \text{LWMC}(C|\sigma_i;\ S_i, W)$
```

Algorithm 2 **LWMCR**(CNF C, substs. S, weights W)

```
// Lifted base case
if all clauses in $C$ are satisfied then
    return $\prod_{A \in \mathbf{A}(C)} (W_A + W_{\neg A})^{n_A(S)}$
if $C$ has an empty unsatisfied clause then return 0
// Lifted decomposition step
if there exists a lifted decomposition $\{C_{1,1}, \ldots, C_{1,m_1},$
    $\ldots, C_{k,1}, \ldots, C_{k,m_k}\}$ of $C$ under $S$ then
    return $\prod_{i=1}^{k} [\text{LWMCR}(C_{i,1}, S, W)]^{m_i}$
// Lifted splitting step
Choose an atom $A$
Let $\{\Sigma_{A,S}^{(1)}, \ldots, \Sigma_{A,S}^{(l)}\}$ be a lifted split of $A$ for $C$ under $S$
if $A$ is defined recursively
    Estimate the probabilities $P_A$ and $P_{\neg A}$ of $A$ being True
    and False respectively using Monte Carlo simulation
    return $\sum_{i=1}^{l} n_i P_A^{t_i} P_{\neg A}^{f_i} \text{LWMCR}(C|\sigma_i;\ S_i, W)$
else
    return $\sum_{i=1}^{l} n_i W_A^{t_i} W_{\neg A}^{f_i} \text{LWMCR}(C|\sigma_i;\ S_i, W)$
```

LWMC is an exact algorithm. However, it lends itself readily to Monte Carlo approximations: we just replace the sum in the splitting step with a random choice of one of its terms, calling the algorithm many times, and averaging the results via importance sampling. We can also improve the performance of LWMC using many optimizations such as unit propagation and caching. Several such optimizations are included in Alchemy 2.0, our open source software that implements PTP.

2.3 Handling Recursive PKBs

A potential problem with using LWMC on recursive PKBs is that the algorithm may not halt. However, we can easily circumvent this problem by modifying LWMC, yielding a new algorithm LWMCR (see Algorithm 2). The key idea here is that while splitting an atom, if we realize that it is recursively defined, we simply replace its True and False weights (W_A and $W_{\neg A}$) by the respective marginal probabilities (P_A and $P_{\neg A}$), estimated using a Monte Carlo simulation technique (e.g., MCMC, rejection sampling, advanced dynamic programming techniques such as in [10], etc.).

References

[1] R. J. Bayardo, Jr. and J. D. Pehoushek. Counting Models Using Connected Components. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pages 157–162, Austin, TX, 2000. AAAI Press.

[2] C. Boutilier, N. Friedman, M. Goldszmidt, and D. Koller. Context-Specific Independence in Bayesian Networks. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence*, pages 115–123, 1996.

[3] M. Chavira and A. Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6-7):772–799, 2008.

[4] P. Domingos and D. Lowd. *Markov Logic: An Interface Layer for Artificial Intelligence*. Morgan & Claypool, San Rafael, CA, 2009.

[5] V. Gogate and P. Domingos. Probabilistic Theorem Proving. In *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence*, pages 256–265. AUAI Press, 2011.

[6] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*, pages 220–229, 2008.

[7] A. Jha, V. Gogate, A. Meliou, and D. Suciu. Lifted Inference from the Other Side: The tractable Features. In *Proceedings of the 24th Annual Conference on Neural Information Processing Systems (NIPS)*, pages 973–981, 2010.

[8] A. Pfeffer. IBAL: A Probabilistic Rational Programming Language. In Bernhard Nebel, editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 733–740. Morgan Kaufmann, 2001.

[9] T. Sang, P. Beame, and H. Kautz. Solving Bayesian networks by weighted model counting. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*, pages 475–482, 2005.

[10] A. Stuhlmüller and N. D. Goodman. A dynamic programming algorithm for inference in recursive probabilistic programs. *Second Statistical Relational AI workshop at UAI 2012 (StaRAI-12)*, 2012.