Anytime Inference in Probabilistic Logic Programs with $T_\mathcal{P}$-Compilation

Jonas Vlasselaer, Guy Van den Broeck, Angelika Kimmig, Wannes Meert, Luc De Raedt
Department of Computer Science
KU Leuven, Belgium
firstname.lastname@cs.kuleuven.be

Abstract

Existing techniques for inference in probabilistic logic programs are sequential: they first compute the relevant propositional formula for the query of interest, then compile it into a tractable target representation and finally, perform weighted model counting on the resulting representation. We propose $T_\mathcal{P}$-compilation, a new inference technique based on forward reasoning. $T_\mathcal{P}$-compilation proceeds incrementally in that it interleaves the knowledge compilation step for weighted model counting with forward reasoning on the logic program. This leads to a novel anytime algorithm that provides hard bounds on the inferred probabilities. Furthermore, an empirical evaluation shows that $T_\mathcal{P}$-compilation effectively handles larger instances of complex real-world problems than current sequential approaches, both for exact and for anytime approximate inference.

1 Introduction

Research on combining probability and logic for use with relational data has contributed many probabilistic logic programming (PLP) languages and systems such as PRISM [Sato, 1995], ICL [Poole, 2008], PITA [Riguzzi and Swift, 2011] and ProbLog [De Raedt *et al.*, 2007; Fierens *et al.*, 2013]. Inference algorithms for PLP often rely on a three step procedure: (1) transform the dependency structure of the logic program and the queries into a propositional formula, (2) compile this formula into a tractable target representation, and (3) compute the weighted model count (WMC) [Chavira and Darwiche, 2008] of the compiled formula. Step (1) is shared by exact inference [Fierens *et al.*, 2013] and approximation techniques based on compiling selected subformulas or sampling [Renkens *et al.*, 2014; Poon and Domingos, 2006]. It is well-known that this step is computationally expensive or even prohibitive for highly cyclic logic programs, as additional propositions are needed to break every cycle [Fierens *et al.*, 2013]. This limits the applicability of the sequential approach in real-world domains with cyclic dependencies, such as gene interaction networks, social networks and the web. The most common solution is to use an approximate (simplified) logic program. Recently, the problem has also been addressed using lazy clause generation [Aziz *et al.*, 2015], but only for exact inference.

The key contribution of this paper is $T_\mathcal{P}$-compilation, a novel inference technique for probabilistic definite clause programs that interleaves construction and compilation of the propositional formula for WMC (steps (1) and (2)). Our second contribution, and formal basis of $T_\mathcal{P}$-compilation, is the $Tc_\mathcal{P}$ operator that generalizes the $T_\mathcal{P}$ operator from logic programming [Van Emden and Kowalski, 1976] to explicitly construct the formula. At any point, the WMC of the current formula provides a lower bound on the true probability, and we thus realize an anytime algorithm.

As in the purely logical setting, forward reasoning with the $Tc_\mathcal{P}$ operator allows one to answer multiple queries in parallel, which is not supported by earlier anytime PLP algorithms based on backward reasoning [Poole, 1993; De Raedt *et al.*, 2007]. Furthermore, forward reasoning naturally handles cyclic dependencies. This avoids the need for additional propositions for breaking cycles and simplifies the compilation step. Finally, our approach is amenable to online inference. That is, when clauses are added to or deleted from the program, $T_\mathcal{P}$-compilation can update the already compiled formulas, which can cause significant savings compared to restarting inference from scratch. While forward reasoning is common in sampling-based inference approaches in probabilistic programming, e.g., [Milch *et al.*, 2005; Goodman *et al.*, 2008; Gutmann *et al.*, 2011], these do not provide guaranteed lower or upper bounds on the probability of the queries.

We obtain an efficient realization of the $Tc_\mathcal{P}$ operator by representing formulas as Sentential Decision Diagrams (SDD) [Darwiche, 2011], which efficiently support incremental formula construction and WMC. While our approach can easily be extended to handle stratified negation, for ease of presentation we focus on definite clause programs which cover real-world applications such as biological and social networks and web-page classification tasks. An empirical evaluation in these domains demonstrates that $T_\mathcal{P}$-compilation outperforms state-of-the-art sequential approaches on these problems with respect to time, space and quality of results.

The paper is organized as follows. We review the necessary background in Section 2. Sections 3 and 4 formally introduce the $Tc_\mathcal{P}$ operator and corresponding algorithms. We discuss experimental results in Section 5 and conclude in Section 6.

2 Background

We review the basics of (probabilistic) logic programming.

2.1 Logical Inference

A *definite clause program*, or *logic program* for short, is a finite set of *definite clauses*, also called *rules*. A definite clause is a universally quantified expression of the form $\mathtt{h} :- \mathtt{b_1}, ..., \mathtt{b_n}$ where $\mathtt{h}$ and the $\mathtt{b_i}$ are atoms and the comma denotes conjunction. The atom $\mathtt{h}$ is called the *head* of the clause and $\mathtt{b_1}, ..., \mathtt{b_n}$ the *body*. A *fact* is a clause that has `true` as its body and is written more compactly as $\mathtt{h}$. If an expression does not contain variables it is *ground*.

Let $\mathcal{A}$ be the set of all ground atoms that can be constructed from the constants, functors and predicates in a logic program $\mathcal{P}$. A *Herbrand interpretation* of $\mathcal{P}$ is a truth value assignment to all $a \in \mathcal{A}$, and is often written as the subset of `true` atoms (with all others being `false`), or as a conjunction of atoms. A Herbrand interpretation satisfying all rules in the program $\mathcal{P}$ is a *Herbrand model*. The model-theoretic semantics of a logic program is given by its unique *least Herbrand model*, that is, the set of all ground atoms $a \in \mathcal{A}$ that are entailed by the logic program, written $\mathcal{P} \models a$.

As running example we use a logic program that models a graph (see Figure 1). The facts represent the edges between two nodes in the graph (we ignore the probabilities at this moment) and the rules define whether there is a path between two nodes. Abbreviating predicate names by initials, the least Herbrand model is given by $\{\mathtt{e}(b,a), \mathtt{e}(b,c), \mathtt{e}(a,c), \mathtt{e}(c,a), \mathtt{p}(b,a), \mathtt{p}(b,c), \mathtt{p}(a,c), \mathtt{p}(c,a), \mathtt{p}(a,a), \mathtt{p}(c,c)\}$.

$0.4 :: \mathtt{edge}(b,a).$ $\quad 0.5 :: \mathtt{edge}(b,c).$
$0.8 :: \mathtt{edge}(a,c).$ $\quad 0.7 :: \mathtt{edge}(c,a).$
$\mathtt{path}(X,Y)$ `:-` $\mathtt{edge}(X,Y).$
$\mathtt{path}(X,Y)$ `:-` $\mathtt{edge}(X,Z), \mathtt{path}(Z,Y).$

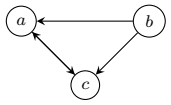


Figure 1: A (probabilistic) logic program modeling a graph.

The task of logical inference is to determine whether a program $\mathcal{P}$ entails a given atom, called *query*. The two most common approaches to inference are *backward reasoning* or *SLD-resolution*, which starts from the query and reasons back towards the facts [Nilsson and Maluszynski, 1995], and *forward reasoning*, which starts from the facts and derives new knowledge using the immediate consequence operator $T_\mathcal{P}$ [Van Emden and Kowalski, 1976].

Definition 1 ($T_\mathcal{P}$ operator) *Let $\mathcal{P}$ be a ground logic program. For a Herbrand interpretation I, the $T_\mathcal{P}$ operator returns*

$$T_\mathcal{P}(I) = \{\mathtt{h} \mid \mathtt{h} \texttt{ :- } \mathtt{b_1}, \ldots, \mathtt{b_n} \in \mathcal{P} \text{ and } \{\mathtt{b_1}, \ldots, \mathtt{b_n}\} \subseteq I\}$$

The least fixpoint of this operator is the least Herbrand model of $\mathcal{P}$. Let $T^k_\mathcal{P}(\emptyset)$ denote the result of k consecutive calls of the $T_\mathcal{P}$ operator, ΔI^i be the difference between $T^{i-1}_\mathcal{P}(\emptyset)$ and $T^i_\mathcal{P}(\emptyset)$, and $T^\infty_\mathcal{P}(\emptyset)$ the least fixpoint interpretation of $T_\mathcal{P}$.

The least fixpoint can be efficiently computed using a seminaive evaluation algorithm [Nilsson and Maluszynski, 1995]. On our example, this results in:

$$\begin{aligned} I^0 &= \emptyset \\ \Delta I^1 &= \{\mathtt{e}(b,a), \mathtt{e}(b,c), \mathtt{e}(a,c), \mathtt{e}(c,a)\} \\ \Delta I^2 &= \{\mathtt{p}(b,a), \mathtt{p}(b,c), \mathtt{p}(a,c), \mathtt{p}(c,a)\} \\ \Delta I^3 &= \{\mathtt{p}(a,a), \mathtt{p}(c,c)\} \\ \Delta I^4 &= \emptyset \end{aligned}$$

$T^\infty_\mathcal{P}(\emptyset) = \bigcup_i \Delta I^i$ is the least Herbrand model as given above.

2.2 Probabilistic-Logical Inference

Most probabilistic logic programming languages (e.g. ICL, PRISM, ProbLog) are based on Sato's *distribution semantics* [Sato, 1995]. In this paper, we use ProbLog as it is the simplest of these languages.

A ProbLog program $\mathcal{P}$ consists of a set $\mathcal{R}$ of *rules* and a set $\mathcal{F}$ of *probabilistic facts*; an example is given in Figure 1. As common, we assume that no probabilistic fact unifies with a rule head. A ProbLog program specifies a probability distribution over its Herbrand interpretations, also called possible worlds. Every grounding $f\theta$ of a probabilistic fact $p :: f$ independently takes value `true` (with probability p) or `false` (with probability $1-p$). For ease of notation, we assume that $\mathcal{F}$ is ground.

A *total choice* $C \subseteq \mathcal{F}$ assigns a truth value to every (ground) probabilistic fact, and the corresponding logic program $C \cup \mathcal{R}$ has a unique least Herbrand model; the probability of this model is that of C. Interpretations that do not correspond to any total choice have probability zero. The probability of a query q is then the sum over all total choices whose program entails q:

$$\Pr(q) := \sum_{C \subseteq \mathcal{F} : C \cup \mathcal{R} \models q} \prod_{f_i \in C} p_i \cdot \prod_{f_i \in \mathcal{F} \setminus C} (1 - p_i) \,. \qquad (1)$$

As enumerating all total choices entailing the query is infeasible, state-of-the-art ProbLog inference [Fierens *et al.*, 2013] reduces the problem to that of weighted model counting. For a formula λ over propositional variables V and a weight function $w(\cdot)$ assigning a real number to every literal for an atom in V, the weighted model count is defined as

$$\mathrm{WMC}(\lambda) := \sum_{I \subseteq V : I \models \lambda} \prod_{a \in I} w(a) \cdot \prod_{a \in V \setminus I} w(\neg a) \,. \qquad (2)$$

The reduction sets $w(f_i) = p_i$ and $w(\neg f_i) = 1 - p_i$ for probabilistic facts $p_i :: f_i$, and $w(a) = w(\neg a) = 1$ else. For a query q, it constructs a formula λ' such that for every total choice $C \subseteq \mathcal{F}$, $C \cup \{\lambda'\} \models q \leftrightarrow C \cup \mathcal{R} \models q$. While λ' may use variables besides the probabilistic facts, their values have to be uniquely defined for each total choice.

We briefly discuss the key steps of the sequential WMC-based approach, and refer to Fierens *et al.* [2013] for full details. First, the *relevant ground program*, i.e., all and only those ground clauses that contribute to some derivation of a query, is obtained using backward reasoning. Next, the ground program is converted into a propositional formula in Conjunctive Normal Form (CNF), which is finally passed to an off-the-shelf solver for weighted model counting.

The resources needed for the conversion step, as well as the size of the resulting CNF, greatly increase with the number of cycles in the ground program, as additional variables and clauses are introduced to capture the least Herbrand semantics of every cycle. For example, the conversion algorithm implemented in ProbLog returns, for the complete grounding of a fully connected graph with only 10 nodes, a CNF with 26995 variables and 109899 clauses.

For exact inference, it is common to compile the CNF into a target representation for which WMC is polynomial in the size of the representation. Anytime (approximate) inference constructs the full CNF first, and then incrementally compiles a set of chosen subformulas [Renkens *et al.*, 2014].

3 The $Tc_{\mathcal{P}}$ Operator

We develop the formal basis of our approach that interleaves formula construction and compilation by means of forward reasoning. The main advantages are that (a) the conversion to propositional logic happens during rather than after reasoning within the least Herbrand semantics, avoiding the expensive introduction of additional variables and propositions, and (b) at any time in the process, the current formulas provide hard bounds on the probabilities.

Although forward reasoning naturally considers all consequences of a program, using the relevant ground program allows us to restrict the approach to the queries of interest. As common in probabilistic logic programming, we assume the *finite support condition*, i.e., the queries depend on a finite number of ground probabilistic facts.

We use forward reasoning to build a formula λ_a for every atom $a \in \mathcal{A}$ such that λ_a exactly describes the total choices $C \subseteq \mathcal{F}$ for which $C \cup \mathcal{R} \models a$. Such λ_a can be used to compute the probability of a via WMC, cf. Section 2.2, independently of their syntactic representation.

Definition 2 (Parameterized interpretation) *A parameterized interpretation $\mathcal{I}$ of a ground probabilistic logic program $\mathcal{P}$ with probabilistic facts $\mathcal{F}$ and atoms $\mathcal{A}$ is a set of tuples (a, λ_a) with $a \in \mathcal{A}$ and λ_a a propositional formula over $\mathcal{F}$.*

For instance, in Figure 1, we can use $\lambda_{e(b,a)} = e(b,a)$ as $e(b,a)$ is `true` in exactly those worlds whose total choice includes that edge, and $\lambda_{p(b,c)} = e(b,c) \vee (e(b,a) \wedge e(a,c))$ as $p(b,c)$ is `true` in exactly those worlds whose total choice includes at least the direct edge or the two-edge path.

A naive approach to construct the λ_a would be to compute $I_i = T^{\infty}_{\mathcal{R} \cup C_i}(\emptyset)$ for every total choice $C_i \subseteq \mathcal{F}$ and to set $\lambda_a = \bigvee_{i:a \in I_i} \bigwedge_{f \in C_i} f$, that is, the disjunction explicitly listing all total choices contributing to the probability of a. Clearly, this requires a number of fixpoint computations exponential in $|\mathcal{F}|$, and furthermore, doing these computations independently does not exploit the potentially large structural overlap between them.

Therefore, we introduce the $Tc_{\mathcal{P}}$ operator. It generalizes the $T_{\mathcal{P}}$ operator to work on the parameterized interpretation and builds, for all atoms in parallel on the fly, formulas that are logically equivalent to the λ_a introduced above. For ease of notation, we assume that every parameterized interpretation implicitly contains a tuple $(\texttt{true}, \top)$, and, just as in regular interpretations, we do not list atoms with $\lambda_a \equiv \bot$. [1]

Definition 3 ($Tc_{\mathcal{P}}$ operator) *Let $\mathcal{P}$ be a ground probabilistic logic program with probabilistic facts $\mathcal{F}$ and atoms $\mathcal{A}$. Let $\mathcal{I}$ be a parameterized interpretation with pairs (a, λ_a). Then, the $Tc_{\mathcal{P}}$ operator is $Tc_{\mathcal{P}}(\mathcal{I}) = \{(a, \lambda'_a) \mid a \in \mathcal{A}\}$ where*

$$\lambda'_a = \begin{cases} a & \text{if } a \in \mathcal{F} \\ \bigvee_{(\texttt{a :- b}_1,\ldots,\texttt{b}_n) \in \mathcal{P}} (\lambda_{b_1} \wedge \cdots \wedge \lambda_{b_n}) & \text{if } a \in \mathcal{A} \setminus \mathcal{F}. \end{cases}$$

Intuitively, where the $T_{\mathcal{P}}$ operator (repeatedly) adds an atom a to the interpretation whenever the body of a rule defining a is `true`, the $Tc_{\mathcal{P}}$ operator adds to the formula for a the description of the total choices for which the rule body is `true`. In contrast to the $T_{\mathcal{P}}$ operator, where a *syntactic* check suffices to detect that the fixpoint is reached, the $Tc_{\mathcal{P}}$ operator requires a *semantic* fixpoint check for each formula λ_a (which we write as $\mathcal{I}^i \equiv Tc_{\mathcal{P}}(\mathcal{I}^{i-1})$).

Definition 4 (Fixpoint of $Tc_{\mathcal{P}}$) *A parameterized interpretation $\mathcal{I}$ is a fixpoint of the $Tc_{\mathcal{P}}$ operator if and only if for all $a \in \mathcal{A}$, $\lambda_a \equiv \lambda'_a$, where λ_a and λ'_a are the formulas for a in $\mathcal{I}$ and $Tc_{\mathcal{P}}(\mathcal{I})$, respectively.*

It is easy to verify that for $\mathcal{F} = \emptyset$, i.e., a ground logic program $\mathcal{P}$, the iterative execution of the $Tc_{\mathcal{P}}$ operator directly mirrors that of the $T_{\mathcal{P}}$ operator, representing atoms as $(a, \top)$.

We use λ^i_a to denote the formula associated with atom a after i iterations of $Tc_{\mathcal{P}}$ starting from $\emptyset$. How to efficiently represent the formulas λ_a is discussed in Section 4.

In our example, the first application of $Tc_{\mathcal{P}}$ sets $\lambda^1_{e(x,y)} = e(x,y)$ for $(x,y) \in \{(b,a),(a,c),(b,c),(c,a)\}$. These remain the same in all subsequent iterations. The second application of $Tc_{\mathcal{P}}$ starts adding formulas for path atoms, which we illustrate for just two atoms:

$$\begin{aligned} \lambda^2_{p(b,c)} &= \lambda^1_{e(b,c)} \vee (\lambda^1_{e(b,a)} \wedge \lambda^1_{p(a,c)}) \vee (\lambda^1_{e(b,c)} \wedge \lambda^1_{p(c,c)}) \\ &= e(b,c) \vee (e(b,a) \wedge \bot) \vee (e(a,c) \wedge \bot) \equiv e(b,c) \\ \lambda^2_{p(c,c)} &= (\lambda^1_{e(c,a)} \wedge \lambda^1_{p(a,c)}) = (e(c,a) \wedge \bot) \equiv \bot \end{aligned}$$

That is, the second step considers paths of length at most 1 and adds $(p(b,c), e(b,c))$ to the parameterized interpretation, but does not add a formula for $p(c,c)$, as no total choices making this atom `true` have been found yet. Similarly, the third iteration adds information on paths of length at most 2:

$$\begin{aligned} \lambda^3_{p(b,c)} &= \lambda^2_{e(b,c)} \vee (\lambda^2_{e(b,a)} \wedge \lambda^2_{p(a,c)}) \vee (\lambda^2_{e(b,c)} \wedge \lambda^2_{p(c,c)}) \\ &\equiv e(b,c) \vee (e(b,a) \wedge e(a,c)) \\ \lambda^3_{p(c,c)} &= (\lambda^2_{e(c,a)} \wedge \lambda^2_{p(a,c)}) \equiv (e(c,a) \wedge e(a,c)) \end{aligned}$$

Intuitively, $Tc_{\mathcal{P}}$ keeps adding longer sequences of edges connecting the corresponding nodes to the path formulas, reaching a fixpoint once all acyclic sequences have been added.

[1] Thus, the empty set implicitly represents the parameterized interpretation $\{(\texttt{true}, \top)\} \cup \{(a, \bot) | a \in \mathcal{A}\}$ for a set of atoms $\mathcal{A}$.

Correctness We now show that for increasing i, $Tc^i_{\mathcal{P}}(\emptyset)$ reaches a least fixpoint where the λ_a are exactly the formulas needed to compute the probability for each atom by WMC.

Theorem 1 *For a ground probabilistic logic program $\mathcal{P}$ with probabilistic facts $\mathcal{F}$, rules $\mathcal{R}$ and atoms $\mathcal{A}$, let λ^i_a be the formula associated with atom a in $Tc^i_{\mathcal{P}}(\emptyset)$. For every atom a, total choice $C \subseteq \mathcal{F}$ and iteration i, we have:*

$$C \models \lambda^i_a \quad \rightarrow \quad C \cup \mathcal{R} \models a$$

Proof by induction: $i = 1$: easily verified. $i \rightarrow i + 1$: easily verified for $a \in \mathcal{F}$; for $a \in \mathcal{A} \setminus \mathcal{F}$, let $C \models \lambda^{i+1}_a$, that is, $C \models \bigvee_{(\texttt{a :- b}_1,\ldots,\texttt{b}_n)\in\mathcal{P}}(\lambda^i_{b_1} \wedge \cdots \wedge \lambda^i_{b_n})$. Thus, there is a $\texttt{a :- b}_1, \ldots, \texttt{b}_n \in \mathcal{P}$ with $C \models \lambda^i_{b_j}$ for all $1 \leq j \leq n$. By assumption, $C \cup R \models b_j$ for all such j and thus $C \cup \mathcal{R} \models a$. □

Thus, after each iteration i, we have $\mathrm{WMC}(\lambda^i_a) \leq \Pr(a)$.

Theorem 2 *For a ground probabilistic logic program $\mathcal{P}$ with probabilistic facts $\mathcal{F}$, rules $\mathcal{R}$ and atoms $\mathcal{A}$, let λ^i_a be the formula associated with atom a in $Tc^i_{\mathcal{P}}(\emptyset)$. For every atom a and total choice $C \subseteq \mathcal{F}$, there is an i_0 such that for every iteration $i \geq i_0$, we have*

$$C \cup \mathcal{R} \models a \quad \leftrightarrow \quad C \models \lambda^i_a$$

Proof: $\leftarrow$: Theorem 1. $\rightarrow$: $C \cup \mathcal{R} \models a$ implies $\exists i_0 \forall i \geq i_0 : a \in T^i_{C\cup\mathcal{R}}(\emptyset)$. We further show $\forall j : a \in T^j_{C\cup\mathcal{R}}(\emptyset) \rightarrow C \models \lambda^j_a$ by induction. $j = 1$: easily verified. $j \rightarrow j + 1$: easily verified for $a \in \mathcal{F}$; for other atoms, $a \in T^{j+1}_{C\cup\mathcal{R}}(\emptyset)$ implies there is a rule $\texttt{a :- b}_1, \ldots, \texttt{b}_n \in \mathcal{R}$ such that $\forall k : b_k \in T^j_{C\cup\mathcal{R}}(\emptyset)$. By assumption, $\forall k : C \models \lambda^j_{b_k}$, and by definition, $C \models \lambda^{j+1}_a$. □

Thus, for every atom a, the λ^i_a reach a fixpoint λ^∞_a exactly describing the possible worlds entailing a, and the $Tc_{\mathcal{P}}$ operator therefore reaches a fixpoint where for all atoms $\Pr(a) = \mathrm{WMC}(\lambda^\infty_a)$.[2] Using Bayes' rule, we can also compute conditional probabilities $\Pr(q|e)$ as $\mathrm{WMC}(\lambda^\infty_q \wedge \lambda'_e)/\mathrm{WMC}(\lambda'_e)$ with $\lambda'_e = \lambda^\infty_e$ for $e = \top$ and $\lambda'_e = \neg\lambda^\infty_e$ for $e = \bot$.

4 Algorithms

As in logical inference with $T_{\mathcal{P}}$, probabilistic inference iteratively calls the $Tc_{\mathcal{P}}$ operator until the fixpoint is reached. This involves incremental formula construction (cf. Definition 3) and equivalence checking (cf. Definition 4). Then, for each query q, the probability is computed as $\mathrm{WMC}(\lambda_q)$.

An efficient realization of our evaluation algorithm is obtained by representing the formulas in the interpretation $\mathcal{I}$ by means of a Sentential Decision Diagram (SDD) [Darwiche, 2011], as these efficiently support all required operations. Hence, we can replace each λ_a in Definition 3 by its equivalent SDD representation (denoted by Λ_a) and each of the *Boolean operations* by the *Apply-operator* for SDDs which, given $\circ \in \{\vee, \wedge\}$ and two SDDs Λ_a and Λ_b, returns an SDD equivalent with $(\Lambda_a \circ \Lambda_b)$.

[2]The finite support condition ensures this happens in finite time.

4.1 $\mathbf{T_{\mathcal{P}}}$-Compilation

The $Tc_{\mathcal{P}}$ operator is, by definition, called on $\mathcal{I}$. To allow for different evaluation strategies, however, we propose a more fine-grained algorithm where, in each iteration, the operator is only called on one specific atom a, i.e., only the rules for which a is the head are evaluated, denoted by $Tc_{\mathcal{P}}(a, \mathcal{I}^{i-1})$. Each iteration i of $T_{\mathcal{P}}$-compilation consists of two steps;

1. Select an atom $a \in \mathcal{A}$.
2. Compute $\mathcal{I}^i = Tc_{\mathcal{P}}(a, \mathcal{I}^{i-1})$

The result of Step 2 is that only the formula for atom a is updated and, for each of the other atoms, the formula in $\mathcal{I}^i$ is the same as in $\mathcal{I}^{i-1}$. It is easy to verify that $T_{\mathcal{P}}$-compilation reaches the fixpoint $Tc^\infty_{\mathcal{P}}(\emptyset)$ in case the selection procedure frequently returns each of the atoms in $\mathcal{P}$.

4.2 Anytime Inference

Until now, we mainly focused on exact inference. Our algorithm is easily adapted for anytime inference as well.

Lower Bound

Following Theorem 1, we know that, after each iteration i, $\mathrm{WMC}(\lambda^i_a)$ is a lower bound on the probability of atom a, i.e. $\mathrm{WMC}(\lambda^i_a) \leq \Pr(a) = \mathrm{WMC}(\lambda^\infty_a)$. To quickly increase $\mathrm{WMC}(\lambda^i_a)$ and, at the same time, avoid a blow-up of the formulas in $\mathcal{I}$, the selection procedure we employ picks the atom which maximizes the following heuristic value:

$$\frac{\mathrm{WMC}(\Lambda^i_a) - \mathrm{WMC}(\Lambda^{i-1}_a)}{\phi_a \cdot (\textsc{Size}(\Lambda^i_a) - \textsc{Size}(\Lambda^{i-1}_a))/\textsc{Size}(\Lambda^{i-1}_a)}$$

where $\textsc{Size}(\Lambda)$ denotes the number of edges in SDD Λ and ϕ_a adjusts for the importance of a in proving queries.

Concretely, Step 1 of $T_{\mathcal{P}}$-compilation calls $Tc_{\mathcal{P}}(a, \mathcal{I}^{i-1})$ for each $a \in \mathcal{A}$, computes the heuristic value and returns the atom a' for which this value is the highest. Then, Step 2 performs $\mathcal{I}^i = Tc_{\mathcal{P}}(a', \mathcal{I}^{i-1})$. Although there is overhead involved in computing the heuristic value, as many formulas are compiled without storing them, this strategy works well in practice.

We take as value for ϕ_a the minimal depth of the atom a in the SLD-tree for each of the queries of interest. This value is a measure for the influence of the atom on the probability of the queries. For our example, and the query $\texttt{p}(b, c)$, the use of ϕ_a would give priority to compile $\texttt{p}(b, c)$ as it is on top of the SLD-tree. Without ϕ_a, the heuristic would give priority to compile $\texttt{p}(a, c)$ as it has the highest probability.

Upper bound

To compute an upper bound, we select $\mathcal{F}' \subset \mathcal{F}$ and treat each $f \in \mathcal{F}'$ as a logical fact rather than a probabilistic fact, that is, we conjoin each λ_a with $\bigwedge_{f\in\mathcal{F}'} \lambda_f$. In doing so, we simplify the compilation step of our algorithm, because the number of possible total choices decreases. Furthermore, at a fixpoint, we have an upper bound on the probability of the atoms, i.e., $\mathrm{WMC}(\lambda^\infty_a|\lambda_{\mathcal{F}'}) \geq \Pr(a)$, because we overestimate the probability of each fact in $\mathcal{F}'$.

Randomly selecting $\mathcal{F}' \subset \mathcal{F}$ does not yield informative upper bounds (they are close to 1). As a heuristic, we compute for each of the facts the minimal depth in the SLD-trees of

the queries of interest and select for $\mathcal{F}'$ all facts whose depth is smaller than some constant d. Hence, we avoid the query to be deterministically `true` as for each of the proofs, i.e., traces in the SLD-tree, we consider at least one probabilistic fact. This yields tighter upper bounds. For our example, and the query $\mathtt{p}(b, c)$, both of the edges starting in node b are at a depth of 1 in the SLD-tree. Hence, it suffices to compile only them, and treat both other edges as logical facts, to obtain an upper bound smaller than 1.

4.3 Online Inference

An advantage of forward reasoning is that it naturally supports online, or incremental, inference. In this setting, one aims to reuse past computation results when making incremental changes to the model, rather than restarting inference from scratch. Our $Tc_{\mathcal{P}}$ operator allows for adding clauses to and removing clauses from the program.

For logic programs, we know that employing the $T_{\mathcal{P}}$ operator on a subset of the fixpoint reaches the fixpoint, i.e. $\forall I \subseteq T^{\infty}_{\mathcal{P}}(\emptyset) : T^{\infty}_{\mathcal{P}}(I) = T^{\infty}_{\mathcal{P}}(\emptyset)$. Moreover, adding definite clauses leads to a superset of the fixpoint, i.e., $T^{\infty}_{\mathcal{P}}(\emptyset) \subseteq T^{\infty}_{P \cup P'}(\emptyset)$. Hence, it is safe to restart the $T_{\mathcal{P}}$ operator from a previous fixpoint after adding clauses. Due to the correspondence established in Theorem 2, this also applies to $Tc_{\mathcal{P}}$. For our example, we could add $0.1 :: \mathtt{e}(a, b)$ and the corresponding ground rules. This leads to new paths, such as $\mathtt{p}(b, b)$, and increases the probability of existing paths, such as $\mathtt{p}(a, c)$.

When removing clauses, atoms in the fixpoint may become invalid. We therefore reset the computed fixpoints for all total choices where the removed clause could have been applied. This is done by conjoining each of the formulas in the parametrized interpretation with the negation of the formula for the head of the removed clause. Then, we restart the $Tc_{\mathcal{P}}$ operator from the adjusted parametrized interpretation, to recompute the fixpoint for the total choices that were removed. For our example, if we remove $0.5 :: \mathtt{e}(b, c)$ and the rules containing this edge, we are removing a possible path from b to c and thus decreasing the probability of $\mathtt{p}(b, c)$.

5 Experimental Results

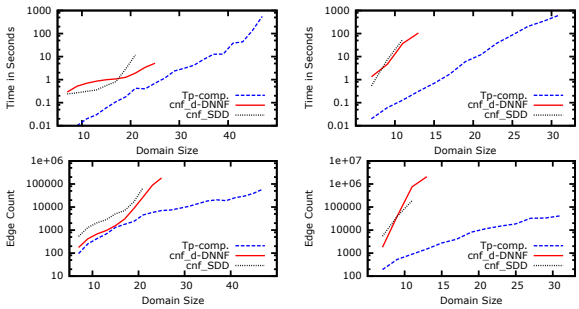
Our experiments address the following questions:

- **Q1** How does $T_{\mathcal{P}}$-compilation compare to exact sequential WMC approaches?
- **Q2** When is online $T_{\mathcal{P}}$-compilation advantageous?
- **Q3** How does $T_{\mathcal{P}}$-compilation compare to anytime sequential approaches?
- **Q4** What is the impact of approximating the model?

We compute relevant ground programs as well as CNFs (where applicable) following Fierens *et al.* [2013] and use the SDD package developed at UCLA[3]. Experiments are run on a 3.4 GHz machine with 16 GB of memory. Our implementation is available as part of the ProbLog package [4].

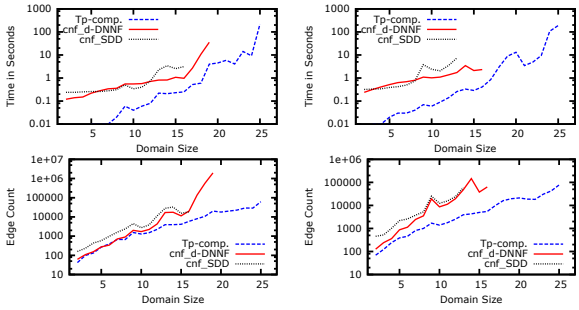
[3] `http://reasoning.cs.ucla.edu/sdd/`

[4] `http://dtai.cs.kuleuven.be/problog/`



One query. Multiple queries.

Figure 2: Exact inference on *Alzheimer*.



One query. Multiple queries.

Figure 3: Exact inference on *Smokers*.

Exact Inference

We use datasets of increasing size from two domains:

Smokers. Following Fierens *et al.* [2013], we generate random power law graphs for the standard 'Smokers' social network domain.

Alzheimer. We use series of connected subgraphs of the Alzheimer network of De Raedt *et al.* [2007], starting from a subsample connecting the two genes of interest 'HGNC 582' and 'HGNC 983', and adding nodes that are connected with at least two other nodes in the graph.

The relevant ground program is computed for one specific query as well as for multiple queries. For the *Smokers* domain, this is `cancer(p)` for a specific person `p` versus for each person. For the *Alzheimer* domain, this is the connection between the two genes of interest versus all genes.

For the sequential approach, we perform WMC using either SDDs, or d-DNNFs compiled with c2d[5] [Darwiche, 2004]. For each domain size (#persons or #nodes) we consider nine instances with a timeout of one hour per setting. We report median running times and target representation sizes, using the standard measure of #edges for the d-DNNFs and $3 \cdot$ #edges for the SDDs. The results are depicted in Figure 2 and 3 and provide an answer for **Q1**.

In all cases, our $T_{\mathcal{P}}$-compilation (`Tp-comp`) scales to larger domains than the sequential approach with both SDD (`cnf_SDD`) and d-DNNF (`cnf_d-DNNF`) and produces smaller compiled structures, which makes subsequent WMC

[5] `http://reasoning.cs.ucla.edu/c2d/`

computations more efficient. The smaller structures are mainly obtained because our approach does not require auxiliary variables to correctly handle the cycles in the program. For *Smokers*, all queries depend on almost the full network structure, and the relevant ground programs – and thus the performance of $T_\mathcal{P}$-compilation – for one or all queries are almost identical. The difference between the settings for the sequential approaches is due to CNF conversion introducing more variables in case of multiple queries.

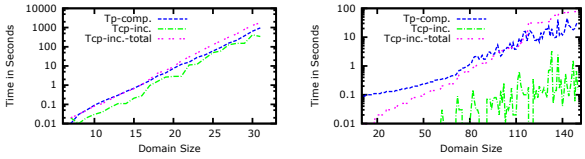


Figure 4: Online inference on *Alzheimer* (left) and *Smokers* (right).

Online Inference

We experiment on the Alzheimer domain discussed above, and a variant on the smokers domain. The smokers network has 150 persons and is the union of ten different random power law graphs with 15 nodes each. We consider the multiple queries setting only, and again report results for nine runs.

We compare our standard $T_\mathcal{P}$-compilation algorithm, which compiles the networks for each of the domain sizes from scratch, with the online algorithm discussed in Section 4.3. The results are depicted in Figure 4 and provide an answer to **Q2**.

For the Alzheimer domain, which is highly connected, incrementally adding the nodes (`Tcp-inc`) has no real benefit compared to recompiling the network from scratch (`Tp-comp`) and, consequently, the cumulative time of the incremental approach (`Tcp-inc-total`) is higher. For the smokers domain, on the other hand, the incremental approach is more efficient compared to recompiling the network, as it only updates the subnetwork to which the most recent person has been added.

Anytime Inference

We consider an approximated ($\mathcal{P}_{apr}$) as well as the original ($\mathcal{P}_{org}$) model of two domains:

Genes. Following Renkens *et al.* [2012; 2014], we use the biological network of Ourfali *et al.* [2007] and its 500 connection queries on gene pairs. The original $\mathcal{P}_{org}$ considers connections of arbitrary length, whereas $\mathcal{P}_{apr}$ restricts connections to a maximum of five edges.

WebKB. We use the WebKB[6] dataset restricted to the 100 most frequent words [Davis and Domingos, 2009] and with random probabilities from $[0.01, 0.1]$. Following Renkens *et al.* [2014], $\mathcal{P}_{apr}$ is a random subsample of 150 pages. $\mathcal{P}_{org}$ uses all pages from the Cornell database. This results in a dataset with 63 queries for the class of a page.

We employ the anytime algorithm as discussed in Section 4.2 and alternate between computations for lower and upper bound at fixed intervals. We compare against two sequential approaches. The first compiles subformulas of the CNF selected by weighted partial MaxSAT (WPMS) [Renkens *et al.*, 2014], the second approximates the WMC of the formula by sampling using the MC-SAT algorithm implemented in the Alchemy package[7].

Following Renkens *et al.* [2014], we run inference for each query separately. The time budget is 5 minutes for $\mathcal{P}_{apr}$ and 15 minutes for $\mathcal{P}_{org}$ (excluding the time to construct the relevant ground program). For MC-SAT, we sample either 5,000 or 10,000 times per variable in the CNF, which yields approximately the same runtime as our approach. Results are depicted in Tables 1, 2 and Figure 5 and allow us to answer **Q3** and **Q4**.

Table 1 shows that $T_\mathcal{P}$-compilation returns bounds for all queries in all settings, whereas WPMS did not produce any answer for $\mathcal{P}_{org}$. The latter is due to reaching the time limit before conversion to CNF was completed. For the approximate model $\mathcal{P}_{apr}$ on the Genes domain, both approaches solve a majority of queries (almost) exactly. Figure 5 plots the number of queries that reached a bound difference below different thresholds against the running time, showing that $T_\mathcal{P}$-compilation converges faster than WPMS. Finally, for the Genes domain, Table 2 shows the number of queries where the result of MC-SAT (using different numbers of samples

		$\mathcal{P}_{apr}$		$\mathcal{P}_{org}$	
		WPMS	$T_\mathcal{P}$-comp	WPMS	$T_\mathcal{P}$-comp
Genes	`Almost Exact`	308	419	0	30
	`Tight Bound`	135	81	0	207
	`Loose Bound`	54	0	0	263
	`No Answer`	3	0	500	0
WebKB	`Almost Exact`	1	7	0	0
	`Tight Bound`	2	34	0	19
	`Loose Bound`	2	22	0	44
	`No Answer`	58	0	63	0

Table 1: Anytime inference: Number of queries with difference between bounds < 0.01 (`Almost Exact`), in $[0.01, 0.25)$ (`Tight Bound`), in $[0.25, 1.0)$ (`Loose Bound`), and 1.0 (`No Answer`).

		$\mathcal{P}_{apr}$		$\mathcal{P}_{org}$
		MCsat_{5000}	MCsat_{10000}	MCsat
Genes	`In Bounds`	150	151	0
	`Out Bounds`	350	349	0
	`N/A`	0	0	500

Table 2: Anytime inference with MC-SAT: numbers of results within and outside the bounds obtained by $T_\mathcal{P}$-compilation on $\mathcal{P}_{apr}$, using 5000 or 10000 samples per CNF variable.

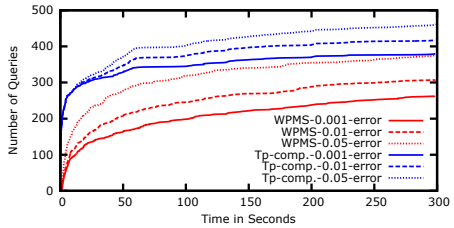


Figure 5: Anytime inference on Genes with $\mathcal{P}_{apr}$: number of queries with bound difference below threshold at any time.

[6] `http://www.cs.cmu.edu/webkb/`

[7] `http://alchemy.cs.washington.edu/`

per variable in the CNF) lies within or outside the bounds computed by $T_{\mathcal{P}}$-compilation. For the original model, no complete CNF is available within the time budget; for the approximate model, more than two thirds of the results are outside the guaranteed bounds obtained by our approach.

We further observed that for 53 queries on the Genes domain, the lower bound returned by our approach using the original model is higher than the upper bound returned by WPMS with the approximated model. This illustrates that computing upper bounds on an approximate model does not provide any guarantees with respect to the full model. On the other hand, for 423 queries in the Genes domain, $T_{\mathcal{P}}$-compilation obtained higher lower bounds with $\mathcal{P}_{apr}$ than with $\mathcal{P}_{org}$, and lower bounds are guaranteed in both cases.

In summary, we conclude that approximating the model can result in misleading upper bounds, but reaches better lower bounds (**Q4**), and that $T_{\mathcal{P}}$-compilation outperforms the sequential approaches for time, space and quality of result in all experiments (**Q3**).

6 Conclusions

We have introduced $T_{\mathcal{P}}$-compilation, a novel anytime inference approach for probabilistic logic programs that combines the advantages of forward reasoning with state-of-the-art techniques for weighted model counting. Our extensive experimental evaluation demonstrates that the new technique outperforms existing exact and approximate techniques on real-world applications such as biological and social networks and web-page classification.

Acknowledgments

We wish to thank Bart Bogaerts for useful discussions. Jonas Vlasselaer is supported by the IWT (agentschap voor Innovatie door Wetenschap en Technologie). Guy Van den Broeck and Angelika Kimmig are supported by the Research Foundation-Flanders (FWO-Vlaanderen).

References

[Aziz *et al.*, 2015] Rehan Abdul Aziz, Geoffrey Chu, Christian Muise, and Peter J. Stuckey. Stable Model Counting and Its Application in Probabilistic Logic Programming. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI)*, 2015.

[Chavira and Darwiche, 2008] Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artif. Intell.*, 172(6-7):772–799, 2008.

[Darwiche, 2004] Adnan Darwiche. New Advances in Compiling CNF into Decomposable Negation Normal Form. In *Proceedings of European Conference on Artificial Intelligence (ECAI)*, pages 328–332, 2004.

[Darwiche, 2011] Adnan Darwiche. SDD: A New Canonical Representation of Propositional Knowledge Bases. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 819–826, 2011.

[Davis and Domingos, 2009] Jesse Davis and Pedro Domingos. Deep Transfer via Second-Order Markov Logic. In *Proceedings of the 26th International Conference on Machine Learning (ICML)*, 2009.

[De Raedt *et al.*, 2007] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)*, 2007.

[Fierens *et al.*, 2013] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *Theory and Practice of Logic Programming*, 2013.

[Goodman *et al.*, 2008] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In *Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence (UAI)*, pages 220–229, 2008.

[Gutmann *et al.*, 2011] Bernd Gutmann, Ingo Thon, Angelika Kimmig, Maurice Bruynooghe, and Luc De Raedt. The magic of logical inference in probabilistic programming. *Theory and Practice of Logic Programming*, 11:663–680, 2011.

[Milch *et al.*, 2005] Brian Milch, Bhaskara Marthi, Stuart J. Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. BLOG: Probabilistic Models with Unknown Objects. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1352–1359, 2005.

[Nilsson and Maluszynski, 1995] Ulf Nilsson and Jan Maluszynski. *Logic, Programming, and PROLOG*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 1995.

[Ourfali *et al.*, 2007] Oved Ourfali, Tomer Shlomi, Trey Ideker, Eytan Ruppin, and Roded Sharan. SPINE: a framework for signaling-regulatory pathway inference from cause-effect experiments. *Bioinformatics*, 23(13):359–366, 2007.

[Poole, 1993] David Poole. Logic programming, abduction and probability. *New Generation Computing*, 11:377–400, 1993.

[Poole, 2008] David Poole. The Independent Choice Logic and Beyond. In L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton, editors, *Probabilistic Inductive Logic Programming*, volume 4911 of *Lecture Notes in Artificial Intelligence*, pages 222–243. Springer, 2008.

[Poon and Domingos, 2006] Hoifung Poon and Pedro Domingos. Sound and Efficient Inference with Probabilistic and Deterministic Dependencies. In *Proceedings of the 21st National Conference on Artificial Intelligence*, pages 458–463, 2006.

[Renkens *et al.*, 2012] Joris Renkens, Guy Van den Broeck, and Siegfried Nijssen. k-optimal: A novel approximate inference algorithm for ProbLog. *Machine Learning*, 89(3):215–231, 2012.

[Renkens *et al.*, 2014] Joris Renkens, Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. Explanation-based approximate weighted model counting for probabilistic logics. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI)*, pages 2490–2496, 2014.

[Riguzzi and Swift, 2011] Fabrizio Riguzzi and Terrance Swift. The PITA System: Tabling and Answer Subsumption for Reasoning under Uncertainty. *Theory and Practice of Logic Programming*, 11(4–5):433–449, 2011.

[Sato, 1995] Taisuke Sato. A statistical learning method for logic programs with distribution semantics. In *Proceedings of the 12th International Conference on Logic Programming (ICLP)*, 1995.

[Van Emden and Kowalski, 1976] Maarten. H. Van Emden and Robert. A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23:569–574, 1976.