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

Jonas Vlasselaer[a], Guy Van den Broeck[b], Angelika Kimmig[a], Wannes Meert[a], Luc De Raedt[a]

[a] *KU Leuven, Belgium*
firstname.lastname@cs.kuleuven.be
[b] *University of California, Los Angeles*
guyvdb@cs.ucla.edu

Abstract

We propose $T_{\mathcal{P}}$-compilation, a new inference technique for probabilistic logic programs that is 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. The main difference with existing inference techniques for probabilistic logic programs is that these are a sequence of isolated transformations. Typically, these transformations include conversion of the ground program into an equivalent propositional formula and compilation of this formula into a more tractable target representation for weighted model counting. An empirical evaluation shows that $T_{\mathcal{P}}$-compilation effectively handles larger instances of complex or cyclic real-world problems than current sequential approaches, both for exact and anytime approximate inference. Furthermore, we show that $T_{\mathcal{P}}$-compilation is conducive to inference in dynamic domains as it supports efficient updates to the compiled model.

Keywords: Probabilistic Inference, Knowledge Compilation, Probabilistic Logic Programs, Dynamic Relational Models

1. Introduction

During the last few years there has been a significant interest in combining relational structure with uncertainty. This has resulted in the fields of statistical relational learning (Getoor and Taskar, 2007; De Raedt *et al.*,

2008), probabilistic programming (Pfeffer, 2014) and probabilistic databases (Suciu *et al.*, 2011), which all address this combination. Probabilistic logic programming (PLP) languages such as PRISM (Sato, 1995), ICL (Poole, 1993), ProbLog (De Raedt *et al.*, 2007), LPADs (Vennekens *et al.*, 2004) and CP-logic (Vennekens *et al.*, 2009) form one stream of work in these fields. These formalisms extend the logic programming language Prolog with *probabilistic* choices on which facts are `true` or `false`. We refer to De Raedt and Kimmig (2015) for a detailed overview.

A common task in PLP is to compute the probability of a set of queries, possibly given some observations. A recent insight is that probabilistic inference for PLP corresponds to weighted model counting (WMC) on weighted logical formulas (Fierens *et al.*, 2015; Chavira and Darwiche, 2008; Darwiche, 2009). As a result, state-of-the-art inference techniques rely on a three step procedure (Fierens *et al.*, 2015): (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). A recurring problem is that the first two steps are computationally expensive, and infeasible for some real-world domains, especially if the domain is highly cyclic.

The key contribution of this paper is $T_\mathcal{P}$-compilation, a novel inference technique for probabilistic logic programs that interleaves construction and compilation of the propositional formula. $T_\mathcal{P}$-compilation uses the $Tc_\mathcal{P}$ operator that generalizes the $T_\mathcal{P}$ operator (Van Emden and Kowalski, 1976) from logic programming to probabilistic logic programming. Whereas the $T_\mathcal{P}$ operator finds one interpretation that satisfies the program, the $Tc_\mathcal{P}$ operator finds all possible interpretations and associated probabilities. At any point, after each application of the $Tc_\mathcal{P}$ operator, the WMC provides a lower bound on the true probability of the queries and we thus realize an anytime algorithm. We obtain an efficient realization of the $Tc_\mathcal{P}$ operator by representing formulas as Sentential Decision Diagrams, which efficiently support incremental formula construction and WMC (Darwiche, 2011).

The main advantage of $T_\mathcal{P}$-compilation is that forward reasoning allows for a natural way to handle cyclic dependencies. Because our approach does not require to convert the complete logic program into a propositional formula before compilation, no additional variables are introduced to break cycles. This considerably simplifies the compilation step and pushes the boundaries for exact as well as approximate inference.

A second advantage is that constructing the formula incrementally results

in a set of formulas, rather than one big formula, which enables program updates and inference in dynamic models, i.e., applications where time is involved. In case of program updates, clauses are added to or deleted from the program and $T_\mathcal{P}$-compilation can update the already compiled formulas. This can cause significant savings compared to restarting inference from scratch. Dynamic models contain time explicitly, introducing a repeated structure in the model. $T_\mathcal{P}$-compilation allows us to exploit this structure, as well as the given observations, to efficiently perform inference.

An empirical evaluation on real-world applications of biological and social networks, web-page classification tasks and games demonstrates how our approach efficiently copes with cyclic and dynamic domains. We show that $T_\mathcal{P}$-compilation outperforms state-of-the-art approaches on these problems with respect to time, space and quality of results.

This paper extends the work of Vlasselaer *et al.* (2015) to stratified probabilistic logic programs with negation. Earlier work of Bogaerts and Van den Broeck (2015) extends the $Tc_\mathcal{P}$ operator towards general logic programs by means of approximation fixpoint theory, which is very general and powerful. When one is only interested in stratified logic programs, however, a much simpler result suffices, which we present here. Furthermore, we show how $T_\mathcal{P}$-compilation allows us to efficiently cope with dynamic models. This dynamic approach is inspired by the work presented in Vlasselaer *et al.* (2016) but differs in two aspects. Firstly, we consider dynamic relational models while Vlasselaer *et al.* (2016) consider dynamic Bayesian Networks, i.e., models without cycles. Secondly, $T_\mathcal{P}$-compilation allows for a more flexible approach that additionally exploits the observations to further scale up inference.

The paper is organized as follows. We start by reviewing the necessary background in Section 2. Sections 3 and 4 formally introduce the $Tc_\mathcal{P}$ operator and corresponding algorithm. We deal with dynamic domains in Section 5 and discuss experimental results in Section 6. Before concluding, we discuss related work in Section 7.

2. Background

We review the basics of (probabilistic) logic programming. First, we introduce definite clause programs and show how to include negation. Next, we discuss probabilistic logic inference and knowledge compilation.

2.1. Logical Inference for Definite Clause Programs

A *term* is a variable, a constant, or a functor applied on terms. An *atom* is of the form $p(t_1, \ldots, t_m)$ where p is a predicate of arity m and the t_i are terms. 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*. The meaning of such a clause is that whenever the body is true, the head has to be true as well. A *fact* is a clause that has `true` as its body and is written more compactly as $\mathtt{h}$. A *definite clause program* is a finite set of definite clauses, also called *rules*. 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 definite clause 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 definite clause 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 $\mathcal{P} \models a$.

Example 1. *Consider the following definite clause program:*

$$\begin{array}{ll} \mathtt{edge}(b, a). & \mathtt{edge}(b, c). \\ \mathtt{edge}(a, c). & \mathtt{edge}(c, a). \\ \mathtt{path}(X, Y) & \mathtt{:-}\ \mathtt{edge}(X, Y). \\ \mathtt{path}(X, Y) & \mathtt{:-}\ \mathtt{edge}(X, Z), \mathtt{path}(Z, Y). \end{array}$$

The facts represent the edges between two nodes in a graph (see Figure 1) and the rules define whether there is a path between two nodes. Abbreviating predicate names by initials, the least Herbrand model is given by $\{\mathtt{e}(b, a)$, $\mathtt{e}(b, c)$, $\mathtt{e}(a, c)$, $\mathtt{e}(c, a)$, $\mathtt{p}(b, a)$, $\mathtt{p}(b, c)$, $\mathtt{p}(a, c)$, $\mathtt{p}(c, a)$, $\mathtt{p}(a, a)$, $\mathtt{p}(c, c)\}$.

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*, and *forward reasoning*. The former starts from the query and reasons back towards the facts (Nilsson and Maluszynski, 1995). For the program depicted in Example 1 and the query

Figure 1: A graph on the left and probabilistic graph on the right.

$\texttt{path}(b, c)$, part of the SLD-tree is shown in Figure 2. The latter starts from the facts and derives new knowledge using the immediate consequence operator $T_{\mathcal{P}}$ (Van Emden and Kowalski, 1976).

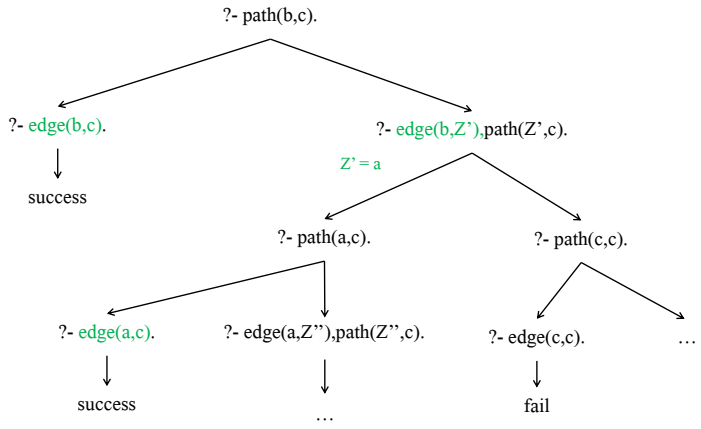


Figure 2: (Part of) SLD-tree for the program shown in Example 1 and the query $\texttt{path}(b, c)$.

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

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

The *least fixpoint* of this operator is the least Herbrand model of $\mathcal{P}$ and is the least set of atoms I such that $T_{\mathcal{P}}(I) \equiv I$. Let $T_{\mathcal{P}}^{k}(\emptyset)$ denote the result of k consecutive calls of the $T_{\mathcal{P}}$ operator, ΔI^i be the difference between $T_{\mathcal{P}}^{i-1}(\emptyset)$ and $T_{\mathcal{P}}^{i}(\emptyset)$. Then $T_{\mathcal{P}}^{\infty}(\emptyset)$ is the least fixpoint interpretation of $T_{\mathcal{P}}$.

Example 2. *The least fixpoint can be computed efficiently using a semi-naive evaluation algorithm (Nilsson and Maluszynski, 1995). For the program*

given in Example 1, this results in:

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

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

Property 1. *Employing the* $T_{\mathcal{P}}$ *operator on a subset of the least fixpoint leads again to the same least fixpoint:*

$$\forall I \subseteq T_{\mathcal{P}}^{\infty}(\emptyset) : T_{\mathcal{P}}^{\infty}(I) = T_{\mathcal{P}}^{\infty}(\emptyset)$$

Property 2. *Adding a set of definite clauses* $\mathcal{P}'$ *to program* $\mathcal{P}$ *leads to a superset of the least fixpoint for* $\mathcal{P}$*:*

$$T_{\mathcal{P}}^{\infty}(\emptyset) \subseteq T_{\mathcal{P} \cup \mathcal{P}'}^{\infty}(\emptyset)$$

2.2. Beyond Definite Clause Programs

A *normal logic program* extends a definite clause program and allows for *negation*, i.e., it is a finite set of *normal clauses* of the form $\mathtt{h} :- \mathtt{b_1}, ..., \mathtt{b_n}$ where $\mathtt{h}$ are atoms and the $\mathtt{b_i}$ are *literals*. A literal is an atom (positive literal) or its negation (negative literal). The $T_{\mathcal{P}}$ operator for definite clause programs can be generalized towards *stratified* normal logic programs, where the rules in the program are partitioned according to their strata (Nilsson and Maluszynski, 1995). Let $\mathcal{P}^{\mathtt{h}}$ be the subset of clauses in $\mathcal{P}$ where $\mathtt{h}$ is the head, then a stratified program is defined as follows:

Definition 2 (Stratified Program). *A normal logic program* $\mathcal{P}$ *is said to be stratified if there exists a partitioning* $\mathcal{P}_1 \cup \cdots \cup \mathcal{P}_m$ *of* $\mathcal{P}$ *such that:*

$$\begin{aligned} &\textit{if } \mathtt{h} \text{ :- } \ldots, \mathtt{b}, \ldots \in \mathcal{P}_i && \textit{then } \mathcal{P}^{\mathtt{b}} \subseteq \mathcal{P}_1 \cup \cdots \cup \mathcal{P}_i \\ &\textit{if } \mathtt{h} \text{ :- } \ldots, \neg\mathtt{b}, \ldots \in \mathcal{P}_i && \textit{then } \mathcal{P}^{\mathtt{b}} \subseteq \mathcal{P}_1 \cup \cdots \cup \mathcal{P}_{i-1} \end{aligned}$$

The $T_{\mathcal{P}}$ operator for normal logic programs is defined as:

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

where $I \models \mathtt{b_i}$ if $\mathtt{b_i} \in I$ and $I \models \neg\mathtt{b_i}$ if $\mathtt{b_i} \notin I$. Then, the *canonical* model[1] can be obtained by iteratively computing the fixpoint for each stratum. Let I_i be the Herbrand interpretation for stratum i, the canonical model for a stratified program with m strata is computed as:

$$\begin{aligned} I_1 &= T^{\infty}_{\mathcal{P}_1}(\emptyset) \\ I_2 &= T^{\infty}_{\mathcal{P}_2}(I_1) \cup I_1 \\ &\vdots \\ I_m &= T^{\infty}_{\mathcal{P}_\mathrm{m}}(I_{m-1}) \cup I_{m-1} \end{aligned}$$

and $T^{\infty}_{\mathcal{P}}(\emptyset) = I_m$.

Example 3. *Consider the following normal logic program with two strata:*

$$\begin{aligned} \mathcal{P}_1 &\left\{ \begin{array}{l} \texttt{sprinklerOn}. \\ \texttt{rain :- cloudy}. \\ \texttt{wetGrass :- rain}. \end{array} \right. \\ \mathcal{P}_2 &\left\{ \begin{array}{l} \texttt{sprinkler :- } \neg\texttt{cloudy}, \texttt{sprinklerOn}. \\ \texttt{wetGrass :- sprinkler}. \end{array} \right. \end{aligned}$$

for which computing the canconical model results in:

$$\begin{aligned} I_1 &= \{\texttt{sprinklerOn}\} \\ I_2 &= \{\texttt{sprinkler}, \texttt{wetGrass}\} \cup I_1 \end{aligned}$$

and $T^{\infty}_{\mathcal{P}}(\emptyset) = \{\texttt{sprinklerOn}, \texttt{sprinkler}, \texttt{wetGrass}\}$.

2.3. Probabilistic Logic Inference

Most probabilistic logic programming languages (e.g., ProbLog, PRISM, ICL) 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*. Without sacrificing generality, 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

[1]For a full discussion of the semantics of general logic programs, we refer to Van Gelder *et al.* (1991).

grounding $f\theta$ of a probabilistic fact $p :: f$ independently takes the value `true` (with probability p) or `false` (with probability $1 - p$). For ease of notation, we assume that $\mathcal{F}$ is ground.

Example 4. *Our program given in Example 1 can be extended with probabilities (see also Figure 1) in the following way:*

$$\begin{aligned}
&0.4 :: \texttt{edge}(b, a). \qquad 0.3 :: \texttt{edge}(b, c).\\
&0.8 :: \texttt{edge}(a, c). \qquad 0.9 :: \texttt{edge}(c, a).\\
&\texttt{path}(X, Y) \texttt{ :- } \texttt{edge}(X, Y).\\
&\texttt{path}(X, Y) \texttt{ :- } \texttt{edge}(X, Z), \texttt{path}(Z, Y).
\end{aligned}$$

The probabilistic facts represent that edges between two nodes are only true with a certain probability. As a consequence, the rules now express a probabilistic path.

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 canonical model (Fierens *et al.*, 2015); 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:

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

As enumerating all total choices entailing the query is infeasible, state-of-the-art ProbLog inference reduces the problem to that of weighted model counting (Fierens *et al.*, 2015). 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

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

The reduction assigns $w(f_i) = p_i$ 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, e.g., variables corresponding to the atoms in the program, their values have to be uniquely defined for each total choice.

Example 5. *Consider the sprinkler program introduced in Example 3 but now,* `sprinklerOn` *and* `cloudy` *are probabilistic facts:*

$$0.7 :: \texttt{sprinklerOn}. \qquad 0.2 :: \texttt{cloudy}.$$

converting the program results in the following propositional formula:

$$\begin{aligned} & \texttt{rain} \leftrightarrow \texttt{cloudy}. \\ \wedge\ & \texttt{sprinkler} \leftrightarrow \neg\texttt{cloudy} \wedge \texttt{sprinklerOn}. \\ \wedge\ & \texttt{wetGrass} \leftrightarrow \texttt{rain} \vee \texttt{sprinkler}. \end{aligned}$$

with $w(\texttt{sprinklerOn}) = 0.7$*,* $w(\neg\texttt{sprinklerOn}) = 0.3$*,* $w(\texttt{cloudy}) = 0.2$ *and* $w(\neg\texttt{cloudy}) = 0.8$*.*

We briefly discuss the key steps of the sequential WMC-based approach, and refer to Fierens *et al.* (2015) for full details. First, the *relevant ground program*, i.e., all and only those ground clauses that contribute to some derivation of a query (and evidence), is obtained using backward reasoning. Next, the ground program is converted to a propositional formula in Conjunctive Normal Form (CNF), which is finally passed to an off-the-shelf solver for weighted model counting.

Example 6. *For the logic program given in Example 1 and a query* $\texttt{p}(\texttt{a}, \texttt{c})$*, we would obtain the following ground program:*

$$\begin{aligned} \texttt{path}(a, c)\ &\texttt{:-}\ \texttt{edge}(a, c). \\ \texttt{path}(a, c)\ &\texttt{:-}\ \texttt{edge}(a, c), \texttt{path}(c, c). \\ \texttt{path}(c, c)\ &\texttt{:-}\ \texttt{edge}(c, a), \texttt{path}(a, c). \end{aligned}$$

For acyclic rules, the conversion step is straightforward and only requires to take *Clark's completion* (Clark, 1978). For programs with cyclic rules, however, the conversion step is more complicated and requires additional variables and clauses to correctly capture the semantics of every cycle. This can be done by rewriting the ground program.

Example 7. *We can correctly break the cycles by rewriting the ground program from Example 6 in the following way:*

$$\begin{aligned} \texttt{path}(a, c)\ &\texttt{:-}\ \texttt{edge}(a, c). \\ \texttt{path}(a, c)\ &\texttt{:-}\ \texttt{edge}(a, c), \texttt{path}(c, c). \\ \texttt{path}(c, c)\ &\texttt{:-}\ \texttt{edge}(c, a), \texttt{aux\_path}(\texttt{a}, \texttt{c}). \\ \texttt{aux\_path}(\texttt{a}, \texttt{c})\ &\texttt{:-}\ \texttt{edge}(a, c). \end{aligned}$$

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. For example, for a fully connected graph with only 10 nodes the conversion algorithm implemented in ProbLog returns a CNF with 26995 variables and 109899 clauses.

Once the ground program is completely converted into a CNF formula, it can be fed to any state-of-the art weighted model counter that supports this standard input format. One of the popular approaches for weighted model counting is supported by knowledge compilation.

2.4. Knowledge Compilation

For probabilistic inference it is common to compile the obtained propositional formula into a target language for which WMC is polynomial in the size of the representation (Darwiche and Marquis, 2002; Chavira and Darwiche, 2008). The most general language that efficiently supports WMC is deterministic decomposable negation normal form (d-DNNF). The properties of d-DNNF state that disjunctions are required to be deterministic, i.e. children of disjunctions cannot be true at the same time, and conjunctions need to be decomposable, i.e. children of conjunctions are not allowed to share variables. Most of the d-DNNF compilers, such as c2d or DSHARP, require a CNF formula as input.

Sentential decision diagram (SDD) (Darwiche, 2011) is a recently introduced target representation that is a strict subset of d-DNNF. An SDD is of the form $(p_1 \wedge s_1) \vee \cdots \vee (p_n \wedge s_n)$ where each pair $(p_i \wedge s_i)$ is called an *element*, the p_i are called *primes* and the s_i are *subs*. Primes and subs are themselves SDDs and the primes are consistent, mutually exclusive and exhaustive. These properties are stronger than the ones for d-DNNF, allowing SDDs to efficiently support Boolean operations on formulas by means of an *apply operator*. More concretely, two SDDs $(p_1 \wedge s_1) \vee \cdots \vee (p_n \wedge s_n)$ and $(p'_1 \wedge s'_1) \vee \ldots (p'_n \wedge s'_n)$ can be combined with a boolean operator $\circ \in \{\vee, \wedge\}$ as follows $(p_1 \wedge p'_1 \wedge s_1 \circ s'_1) \vee \cdots \vee (p_n \wedge p'_n \wedge s_n \circ s'_n)$, only requiring linear time and space. Moreover, compressed SDDs are canonical leading to a practical efficient incremental compilation when performing a large number of recursive apply operations (Van den Broeck and Darwiche, 2015; Choi *et al.*, 2013). Hence, SDDs allow to be efficiently compiled in an incremental (bottom-up) way and do not strictly require to first construct a formula in CNF.

To conclude, we note that ordered binary decision diagram (OBDD) is a subset of SDD and also efficiently supports an apply operator. The advantage of SDD is that it comes with size upper bounds based on treewidth which are tighter than the size upper bounds based on pathwidth for OBDDs.

3. The $T_{C_\mathcal{P}}$ Operator

We now develop the formal basis of our approach. For ease of notation, we first only consider definite clause programs and treat normal logic programs towards the end of this section.

3.1. Definite Clause Programs

The incentive of our inference algorithm is to interleave formula construction and compilation by means of forward reasoning. The two advantages are that (a) the conversion to propositional logic happens during rather than after reasoning within the logic programming 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 2.3).

Definition 3 (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 $\mathcal{F}$ expressing in which interpretations a is true.*

Example 8. *For the program shown in Example 4, the parameterized interpretation is:*

$$\{\big(\mathtt{e}(b,a), \lambda_{\mathtt{e}(b,a)}\big), \big(\mathtt{e}(b,c), \lambda_{\mathtt{e}(b,c)}\big), \big(\mathtt{e}(a,c), \lambda_{\mathtt{e}(a,c)}\big), \big(\mathtt{e}(c,a), \lambda_{\mathtt{e}(c,a)}\big),$$
$$\big(\mathtt{p}(b,a), \lambda_{\mathtt{p}(b,a)}\big), \big(\mathtt{p}(b,c), \lambda_{\mathtt{p}(b,c)}\big), \big(\mathtt{p}(a,c), \lambda_{\mathtt{p}(a,c)}\big), \big(\mathtt{p}(c,a), \lambda_{\mathtt{p}(c,a)}\big),$$
$$\big(\mathtt{p}(a,a), \lambda_{\mathtt{p}(a,a)}\big), \big(\mathtt{p}(c,c), \lambda_{\mathtt{p}(c,c)}\big)\}.$$

and, for instance, $\lambda_{e(b,a)} = e(b,a)$ *since* $e(b,a)$ *is* `true` *in exactly those worlds where the total choice includes this edge, and* $\lambda_{p(b,c)} = e(b,c) \vee [e(b,a) \wedge e(a,c)]$ *since* $p(b,c)$ *is* `true` *in exactly those worlds where the total choice includes the direct edge or the two-edge path over* a*.*

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

Definition 4 ($Tc_{\mathcal{P}}$ **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 5 (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_a^i to denote the formula associated with atom a after i iterations of $Tc_{\mathcal{P}}$ starting from $\emptyset$. We use SDDs to efficiently represent the formulas λ_a as will be discussed in Section 4.

Example 9. *Applying $Tc_{\mathcal{P}}$ to the program given in Example 4 results in the following sequence:*

The first application of $Tc_{\mathcal{P}}$ sets:

$$\lambda^1_{e(b,a)} = e(b,a), \quad \lambda^1_{e(a,c)} = e(a,c), \quad \lambda^1_{e(b,c)} = e(b,c), \quad \lambda^1_{e(c,a)} = e(c,a)$$

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.*

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.

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

Theorem 1. *For a ground probabilistic definite clause program $\mathcal{P}$ with probabilistic facts $\mathcal{F}$, rules $\mathcal{R}$ and atoms $\mathcal{A}$, let λ_a^i 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 $C \cup \mathcal{R} \models a$. □

Corollary 1. *After each iteration i, we have* $\mathrm{WMC}(\lambda^i_a) \leq \Pr(a)$.

Theorem 2. *For a ground probabilistic definite clause 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]

Conditional Probabilities. Once a fixpoint is reached, conditional probabilities can be computed using Bayes' rule. The probability of a query q, given a set $\mathbf{E}$ of observed atoms (*evidence atoms*) and a vector $\mathbf{e}$ of observed truth values, is computed as:

$$\Pr(q|\mathbf{E} = \mathbf{e}) = \frac{\mathrm{WMC}(\lambda^\infty_q \wedge \lambda^\infty_{\mathbf{e}})}{\mathrm{WMC}(\lambda^\infty_{\mathbf{e}})} \qquad \text{with} \quad \lambda^\infty_{\mathbf{e}} = \bigwedge_{e\in\mathbf{e}} \lambda^\infty_e$$

Hence, computing conditional probabilities additionally requires the construction of a formula $\lambda^\infty_{\mathbf{e}}$ that represents the evidence $\mathbf{e}$.

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

3.2. Normal Logic Programs

The correspondence with the $T_\mathcal{P}$ operator allows us to extend the $Tc_\mathcal{P}$ operator towards stratified normal logic programs (Sec. 2.2). To do so, we apply the $Tc_\mathcal{P}$ operator stratum by stratum, that is, we first have to reach a fixpoint for $\mathcal{P}_i$ before considering the rules in $\mathcal{P}_{i+1}$. Let $\mathcal{I}_i$ be the parametrized interpretation for stratum i, the set of formulas for a stratified program with m strata is obtained by:

$$\begin{aligned}\mathcal{I}_1 &= Tc^\infty_{\mathcal{P}_1}(\emptyset)\\ \mathcal{I}_2 &= Tc^\infty_{\mathcal{P}_2}(\mathcal{I}_1)\\ &\vdots\\ \mathcal{I}_m &= Tc^\infty_{\mathcal{P}_\mathrm{m}}(\mathcal{I}_{m-1})\end{aligned}$$

Following the definition of stratified programs, the formula for a negative literal $\neg a$ is only required once a fixpoint for the positive literal a has been reached. Hence, $\lambda_{\neg a}$ can be obtained as $\neg\lambda_a$.

4. Anytime Inference

Probabilistic inference iteratively calls the $Tc_\mathcal{P}$ operator until the fixpoint is reached. This involves incremental formula construction (cf. Definition 4) and equivalence checking (cf. Definition 5).

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), which can handle the required operations efficiently (Darwiche, 2011). Hence, we can replace each λ_a in Definition 4 by its equivalent SDD representation 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 $(\Lambda_a \circ \Lambda_b)$.

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 $Tc_\mathcal{P}(a, \mathcal{I}^{i-1})$. Note that, in case of normal logic programs, we only consider the rules within one stratum. 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 $Tc_{\mathcal{P}}^{\infty}(\emptyset)$ in case the selection procedure frequently returns each of the atoms in $\mathcal{P}$.

4.2. Lower Bound

Following Theorem 1, we know that, after each iteration i, $\text{WMC}(\lambda_a^i)$ is a lower bound on the probability of atom a, i.e., $\text{WMC}(\lambda_a^i) \leq \Pr(a) = \text{WMC}(\lambda_a^{\infty})$, which holds for definite clause programs as well as for stratified normal logic programs. To quickly increase $\text{WMC}(\lambda_a^i)$ 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:

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

where $\textsc{Size}(\mathit{\Lambda})$ denotes the number of edges in SDD $\mathit{\Lambda}$ 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 $\mathtt{p}(b, c)$, the use of ϕ_a would give priority to compile $\mathtt{p}(b, c)$ as it is on top of the SLD-tree (see Figure 2). Without ϕ_a, the heuristic would give priority to compile $\mathtt{p}(a, c)$ as it has the highest probability.

4.3. Upper bound

To compute an upper bound for definite clause programs, 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., $\mathrm{WMC}(\lambda_a^\infty|\lambda_{\mathcal{F}'}) \geq \Pr(a)$, 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 (see Figure 2). Hence, it suffices to compile only them, and treat both other edges as logical facts, to obtain an upper bound smaller than 1.

Discussion. Computation of the lower bounds is also valid for programs with negation but rules need to be compiled according to their stratification. Computation of our upper bounds only holds for definite clause programs, i.e., programs without negation, and is similar in spirit to the upper bounds of Poole (1993) and De Raedt *et al.* (2007). As computation of the lower and upper bound operates on different formulas, an anytime algorithm should alternate between compiling these formulas.

5. Extensions of Tp-Compilation

Modeling real-world applications might require the inclusion of time in an implicit way, e.g. with program updates, or explicit way, e.g. dynamic models. We now show how $T_\mathcal{P}$-compilation allows us to deal efficiently with these models.

5.1. Inference with Program Updates

A probabilistic logic program typically represents a stationary model, that is, the set of rules $\mathcal{R}$ as well as the set of facts $\mathcal{F}$ is fixed. For many applications, however, the program only expresses all available information at one specific moment in time while the underlying process constantly evolves. As an example one can think of a probabilistic graph, as shown in Figure 1, where new edges and nodes are discovered over time, while others disappear.

To cope with new information, the set of facts $\mathcal{F}$ is updated while the set of (non-ground) rules $\mathcal{R}$ remains unchanged. Adding facts leads to new clauses in the ground program and removing facts removes clauses from the

grounded program. One way to deal with program changes is to recompile the complete program from scratch after each update of $\mathcal{F}$. In case $\mathcal{F}$ changes only marginally, however, a more efficient approach is to reuse past compilation results. This can be achieved using the $Tc_{\mathcal{P}}$ operator, which allows for adding clauses to and removing clauses from the program. We only consider program updates for definite clause programs as for normal logic programs we will have to restart compilation from the lowest strata affected by the update.

Adding Clauses. For definite clause programs, we know that employing the $T_{\mathcal{P}}$ operator on a subset of the fixpoint reaches the fixpoint (see Property 1). Moreover, adding definite clauses leads to a superset of the fixpoint (see Property 2). Hence, it is safe to restart the $T_{\mathcal{P}}$ operator from a previous fixpoint after adding clauses. Assume the set of facts $\mathcal{F}$ is extended to $\mathcal{F}'$, then we have $T^{\infty}_{\mathcal{R}\cup\mathcal{F}'}(\emptyset) = T^{\infty}_{\mathcal{R}\cup\mathcal{F}'}(T^{\infty}_{\mathcal{R}\cup\mathcal{F}}(\emptyset))$. 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 $\mathtt{p}(a, c)$.

Removing Clauses. When removing clauses from a definite clause program, 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. Let $\mathcal{I}$ denote the parametrized interpretation and $\mathcal{H}$ the set of atoms in the head of a removed clause, the new parameterized interpretation $\mathcal{I}'$ is obtained as:

$$\mathcal{I}' = \{(a, \lambda'_a) \mid (a, \lambda_a) \in \mathcal{I} \text{ with } \lambda'_a = \lambda_a \wedge \bigwedge_{h \in \mathcal{H}} \neg\lambda_h\}$$

Then, we restart the $Tc_{\mathcal{P}}$ operator from the adjusted parametrized interpretation $\mathcal{I}'$, 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 $\mathtt{p}(b, c)$.

5.2. Inference in Dynamic Models

Probabilistic logic programs allow us to represent stochastic processes by means of a dynamic model (Nitti *et al.*, 2013; Thon *et al.*, 2011). To do so, one explicitly inserts time such that each rule is of the form $\mathtt{h_t} :- \mathtt{b_{1,t1}}, ..., \mathtt{b_{n,tn}}$, where $\mathtt{t}, \mathtt{t1}, \ldots, \mathtt{tn}$ denotes the time step to which the corresponding atom belongs. In this paper, we assume that the first-order Markov property holds, allowing us to rewrite the rules as $\mathtt{h_t} :- \mathtt{b_{1,t}}, \ldots, \mathtt{b_{m,t}}, \mathtt{b_{m+1,t-1}}, \ldots \mathtt{b_{n,t-1}}$, i.e., each atom can only depend on atoms from the same or previous time step. Note that this also implies that cycles cannot range over different time steps.

Example 10. *Consider the following probabilistic logic program*[3] *representing a dynamic social network:*

$$
\begin{aligned}
&0.2 :: \mathtt{coldOutside}_t.\\
&0.4 :: \mathtt{staySick}(X)_t.\\
&\qquad \mathtt{sick}(X)_t \ \mathtt{:-}\ \mathtt{coldOutside}_t.\\
&\qquad \mathtt{sick}(X)_t \ \mathtt{:-}\ \mathtt{friends}(X,Y), \mathtt{sick}(Y)_t.\\
&\qquad \mathtt{sick}(X)_t \ \mathtt{:-}\ \mathtt{sick}(X)_{t-1}, \mathtt{staySick}(X)_t.
\end{aligned}
$$

We assume the `friends` *relation remains constant over time, and therefore drop the time index for these atoms.*

In practice, one distinguishes an initial model from the transition model. The former expresses the *prior distribution*, i.e., the distribution for the first time step, while the latter latter serves as a template for all subsequent time-steps. As the transition model is the one that repeats over time, we focus on the latter.

Inference. One way to perform inference in a dynamic model is by means of *unrolling*, i.e. one grounds the program for a set of queries and runs a standard inference algorithm. It is well-known, however, that this approach scales poorly when inference is required for a large number of time steps. A more efficient alternative is obtained by exploiting the repeated structure that is inherently present in dynamic models (Murphy, 2002). We shortly review the key ideas of inference in dynamic models and refer to Murphy (2002) and Vlasselaer *et al.* (2016) for more details.

[3]The model can be represented as a PLP by including for each atom a time argument T.

The most common inference task in dynamic models is that of *filtering*. Here, the goal is to compute the probability of a query q at time step t given evidence (observations) up to time step τ, i.e., one aims to compute $\Pr(q_t|\mathbf{e}_{1:\tau})$ with $\tau \leq t$. This task can be performed efficiently by repeatedly computing the *forward message* for each time step t. The forward message is the joint probability distribution over all atoms that d-separate the past from the future, i.e., knowing the joint distribution over all these atoms makes the future independent of the past. The set of atoms over which the joint distribution has to be computed is referred to as the *interface*, which we denote as $\mathbf{I}_t$. The forward message can be computed recursively as follows:

$$\Pr(\mathbf{I}_t|\mathbf{e}_{1:t}) = \sum_{\mathbf{i}_{t-1} \in \mathbf{I}_{t-1}} \Pr(\mathbf{I}_t|\mathbf{i}_{t-1}, \mathbf{e}_t) \Pr(\mathbf{i}_{t-1}|\mathbf{e}_{1:t-1}) \tag{3}$$

with $\mathbf{e}_t$ the truth values of the observed atoms at time t and $\mathbf{i}_{t-1}$ one truth-value assignment to atoms in $\mathbf{I}_{t-1}$. Once we have $\Pr(\mathbf{I}_t|\mathbf{e}_{1:t})$, we can compute $\Pr(q_{t+1}|\mathbf{e}_{1:t+1})$.

Example 11. *For the dynamic model presented in Example 10 and a domain of three persons,* `ann`*,* `bob` *and* `cin`*, the interface is the following set of three atoms:*

$$\mathbf{I}_t = \{\texttt{sick(ann)}_t, \texttt{sick(bob)}_t, \texttt{sick(cin)}_t\}.$$

The forward message involves computing the probability of each possible value assignment of the interface given the previous interface and evidence:

$$\begin{aligned}
&\Pr\left(\texttt{sick(ann)}_t \wedge \texttt{sick(bob)}_t \wedge \texttt{sick(cin)}_t \mid \mathbf{I}_{t-1}, \mathbf{e}_t\right),\\
&\Pr\left(\texttt{sick(ann)}_t \wedge \texttt{sick(bob)}_t \wedge \neg\texttt{sick(cin)}_t \mid \mathbf{I}_{t-1}, \mathbf{e}_t\right),\\
&\ldots\\
&\Pr\left(\neg\texttt{sick(ann)}_t \wedge \neg\texttt{sick(bob)}_t \wedge \neg\texttt{sick(cin)}_t \mid \mathbf{I}_{t-1}, \mathbf{e}_t\right)
\end{aligned}$$

Dynamic $T_\mathcal{P}$-compilation. We now extend our $T_\mathcal{P}$-compilation approach to efficiently perform inference in dynamic domains. In doing so, we can distinguish the following two phases:

- *Offline phase*: Use $T_\mathcal{P}$-compilation to compile the transition model. The resulting parametrized interpretation $\mathcal{I}$ contains a tuple (a, λ_a) for each atom a in the transition model. The formulas λ_a will then serve as a template for each time-step that inference is required. The offline phase only has to be performed once.

- *Online phase*: Iterate over time-steps t and compute the forward message. This includes extending the template formulas Λ_a to adjust for the situation, i.e. the evidence, at time t.

Computation of the forward message at time t requires extending the template formulas Λ_a to take into account the evidence for time-step t, denoted with $\mathbf{e}_t$, as well as the distribution from $t-1$. We use Λ_a^t to denote the adjusted formula of atom a for time step t. Let $\mathbf{i}_t$ be one truth-value assignment to atoms in $\mathbf{I}_t$, its conditional probability is computed as:

$$\Pr(\mathbf{i}_t|\mathbf{e}_{1:t}) = \frac{\mathrm{WMC}\left(\bigwedge_{i\in\mathbf{i}_t}\Lambda_i^t\right)}{\mathrm{WMC}(\Lambda_{\mathbf{e}_t}^t)} \tag{4}$$

with

$$\Lambda_i^t = \bigvee_{\mathbf{i}^j\in\mathbf{I}} \left((\Lambda_i \wedge \Lambda_{\mathbf{e}_t})|\mathbf{i}^j\right) \wedge state_{t-1}^j \wedge \mathbf{i}^j \tag{5}$$

where $\Lambda_{\mathbf{e}_t}$ is the formula representing the evidence, as defined before, and $\Lambda|\mathbf{i}$ denotes that the formula Λ is conditioned on the values in $\mathbf{i}$. *Conditioning* transforms the formula by replacing all variables $\mathbf{V}$ in Λ with their assignment in $\mathbf{v}$ and propagates these values while preserving the properties of the formula (Darwiche and Marquis, 2002). The auxiliary variable $state_{t-1}^j$ is required to correctly include the distribution from $t-1$ (cf. Equation 3) and the weight function sets $w(state_{t-1}^j) = \Pr(\mathbf{i}_{t-1}^j|\mathbf{e}_{1:t-1})$. The forward message (the complete distribution) is obtained by repeating Equation 4 for each $\mathbf{i}_t \in \mathbf{I}_t$.

To push more of the computational effort towards the offline phase, we could rewrite Equation 5 as:

$$\Lambda_i^t = \bigvee_{\mathbf{i}^j\in\mathbf{I}} \left((\Lambda_i|\mathbf{i}^j) \wedge \left(\Lambda_{\mathbf{e}_t}|\mathbf{i}^j\right) \wedge state_{t-1}^j \wedge \mathbf{i}^j \right) \tag{6}$$

such that $\Lambda_i|\mathbf{i}^j$ has to be computed only once.

Dealing with Evidence. The key difference of dynamic $T_{\mathcal{P}}$-compilation compared to the *structural interface algorithm* (SIA) introduced by Vlasselaer *et al.* (2016) is how evidence is treated. The incentive of SIA is to push as much of the computational overhead, i.e., all compilation steps, into the offline phase. This requires the compilation of one large formula that contains a variable for each of the atoms in the transition model, including the

ones that are (potentially) observed. Then, the forward message is computed by accordingly setting weights and requires an exponential number of WMC calls.

With our $T_\mathcal{P}$-compilation approach, on the other hand, we do not include evidence by setting weights but adjust the template formulas Λ_a to explicitly contain the evidence. This requires, for each time step t, to conjoin the template formulas with $\Lambda_{\mathbf{e}_t}$, that is, the formula representing the evidence (see Equation 5). Although this approach requires an additional compilation step within the *online phase*, which might be computationally expensive, representing the evidence explicitly allows us to exploit local structure, in the form of determinism, in the forward message. Indeed, if $w(state^j_{t-1}) = 0$ or if $(\Lambda_i \wedge \Lambda_{\mathbf{e}_t}) \equiv \bot$, Equation 5 simplifies significantly. Hence, we avoid an exponential representation of the forward message in case evidence introduces determinism in $\Pr(\mathbf{I}_t|\mathbf{e}_{1:t})$.

Pushing $\Lambda_i|\mathbf{i}^j$ towards the offline phase, as done by Equation 6, still requires an exponential representation and does not maximally exploit potential determinism in the forward message. Hence, Equation 6 is only beneficial in case there is not enough determinism to compensate for the computational overhead in Equation 5. We refer to Equation 5 as *flexible* and to Equation 6 as *fixed* dynamic $T_\mathcal{P}$ compilation.

Discussion. The above introduced approach only considers exact inference for dynamic domains and is, in worst-case, exponential in the number of atoms in the interface. A topic of future work is to investigate how we can extend this approach towards bounded approximate inference as presented by Boyen and Koller (1998).

6. Experimental Results

Our experiments address the following questions:

Q1 How does $T_\mathcal{P}$-compilation compare to exact sequential WMC approaches?

Q2 How does $T_\mathcal{P}$-compilation compare to anytime sequential approaches?

Q3 What is the impact of approximating the model?

Q4 Can we efficiently deal with program updates?

Q5 Can we efficiently exploit evidence in dynamic domains?

We compute relevant ground programs as well as CNFs (where applicable) following Fierens *et al.* (2015) and use the SDD package developed at UCLA[4]. Experiments are run on a 3.4 GHz machine with 16 GB of memory. Our implementation is available as part of the ProbLog package[5].

6.1. Exact Inference

We use datasets of increasing size from two domains:

Smokers. Following Fierens *et al.* (2015), we generate random power law graphs for the standard 'Smokers' social network domain. Cycles in the program are introduced by the following rule:

$$\texttt{smokes}(X) \texttt{ :- } \texttt{friends}(X, Y), \texttt{smokes}(Y).$$

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 logic program (i.e., the set of rules) is similar to the one we used in our example (see Figure 1) and cycles are introduced by the second rule for `path`.

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.

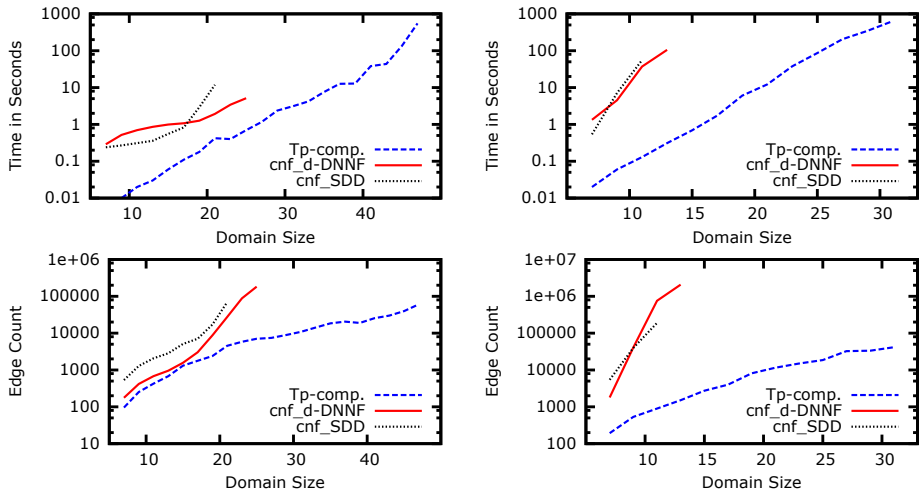
For the sequential approach, we perform WMC using either SDDs, or d-DNNFs compiled with c2d[6] (Darwiche, 2004). For each domain size (#persons or #nodes) we consider nine instances with a timeout of one hour per setting. We report median running times and target representation sizes, using the standard measure of #edges for the d-DNNFs and $3 \cdot$ #edges for the SDDs. The results are depicted in Figure 3 and 4 and provide an answer for **Q1**.

In all cases, the $T_{\mathcal{P}}$-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 computations more efficient. The smaller structures are mainly obtained because our approach does not require auxiliary variables to correctly handle

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

[5] `https://dtai.cs.kuleuven.be/problog/`

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



One query. Multiple queries.

Figure 3: Exact inference on *Alzheimer*.

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.

6.2. 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 logic program is similar to the one we used in our example (see Figure 1). 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[7] dataset restricted to the 100 most frequent words (Davis and Domingos, 2009) and with random probabilities from $[0.01, 0.1]$. Cycles in the program are introduced by the following rule:

$$\texttt{hasClass}(P, C) \texttt{ :- } \texttt{linksTo}(P, P2), \texttt{hasClass}(P2, C2).$$

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

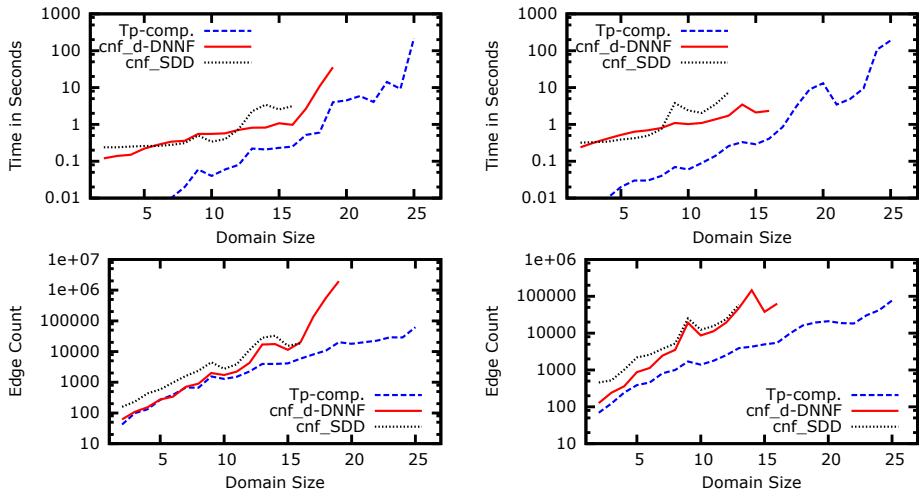


Figure 4: Exact inference on *Smokers*.

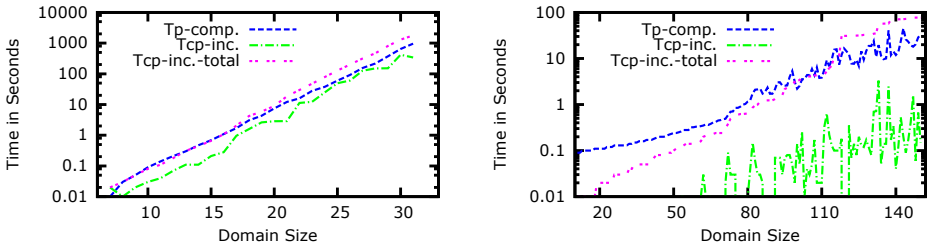


Figure 5: Program updates for *Alzheimer* (left) and *Smokers* (right).

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 Sections 4.2 and 4.3 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[8].

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

		$\mathcal{P}_{apr}$		$\mathcal{P}_{org}$	
		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.

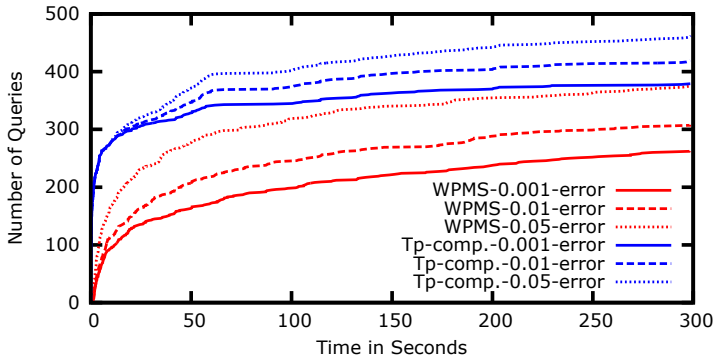


Figure 6: Anytime inference on Genes with $\mathcal{P}_{apr}$: number of queries with bound difference below threshold at any time.

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 6 and allow us to answer **Q2** and **Q3**.

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 6 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 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 (**Q3**), and that $T_{\mathcal{P}}$-compilation outperforms the sequential approaches for time, space and quality of result in all experiments (**Q2**).

6.3. Program Updates

We perform experiments 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 5.1. The results are depicted in Figure 5 and provide an answer to **Q4**.

For the Alzheimer domain, which is highly connected, incrementally adding the nodes (`Tcp-inc`) has no real benefit compared to recompiling the net-

work 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.

6.4. Dynamic Inference

We evaluate dynamic $T_\mathcal{P}$ compilation on two different domains:

Mastermind. Following Vlasselaer *et al.* (2016) we represent the *mastermind game* (Chavira *et al.*, 2006) as a dynamic model where each of the time steps corresponds to one round of playing the game. The goal of the game is to guess the colors of the pegs hidden by the opponent after which the opponent gives feedback whether you guessed the correct colors of the pegs. For each round, we randomly guess the colors of the hidden pegs and feedback is returned by the opponent. The goal is to compute the belief state of the colors of the hidden pegs after 8 rounds of the game. We vary the number of colors (C) as well as the number of pegs (P).

Sickness. We use the dynamic *sickness* domain as depicted in Example 10 with random power law graphs to represent the networks. For each time step, evidence is randomly generated for $x\%$ of the `sick` atoms. The goal is to compute the belief state of the persons being sick after 10 time steps.

We consider nine instances with a timeout of one hour and report median results. For the *mastermind* domain, we compare *fixed* as well as *flexible* dynamic $T_\mathcal{P}$ compilation (cf. Section 5.2) with the *structural interface algorithm* (SIA) (Vlasselaer *et al.*, 2016). The latter is a state-of-the-art inference technique for dynamic Bayesian networks based on d-DNNF compilation. For the *sickness domain*, we compare *flexible* dynamic $T_\mathcal{P}$ compilation for three different levels of evidence, being on 0%, 33% and 66% of the `sick` atoms. The results are depicted in Table 3 and 4 and allow us to answer **Q5**.

For the mastermind game we observe that dynamic $T_\mathcal{P}$ compilation offers significant speed-ups and scales to more complex domains compared to SIA. Furthermore, the flexible approach compares favourable to the fixed approach as compile times are lower and sizes of the obtained representations are smaller. Results for the sickness network show that inference (done with our flexible approach) benefits from exploiting evidence. With the fixed approach we would only get as far as 0% evidence and, although not defined for cyclic programs, SIA would be comparable to 0% evidence for compilation.

Model	SIA size #edges	SIA $T_{offline}$ (s)	SIA T_{inf} (s)	fixed $T_{\mathcal{P}}$ size #edges	fixed $T_{\mathcal{P}}$ $T_{offline}$ (s)	fixed $T_{\mathcal{P}}$ T_{inf} (s)	flexible $T_{\mathcal{P}}$ size #edges	flexible $T_{\mathcal{P}}$ $T_{offline}$ (s)	flexible $T_{\mathcal{P}}$ T_{inf} (s)
C-P	×1000			×1000			×1000		
6 - 3	24	1.3	0.02	84	0.9	0.06	48	0.03	0.06
9 - 3	88	4.9	0.1	345	7.1	0.5	171	0.1	0.6
6 - 4	361	55.2	1.2	741	10.8	0.7	504	0.8	0.6
8 - 4	1,350	220.7	13.6	2,604	60.1	4.1	1,374	3.2	3.2
9 - 4	-	-	-	4,506	130.2	9.8	2,826	5.8	10.1
10 - 4	-	-	-	6,939	281.0	19.2	4,368	10.2	21.6
11 - 4	-	-	-	-	-	-	6,459	17.7	33.5
4 - 5	519	128.6	1.7	657	6.6	0.4	525	1.1	0.2
5 - 5	-	-	-	2,289	30.6	2.0	1,833	4.6	1.5
6 - 5	-	-	-	6,414	111.4	9.6	5,016	15.3	6.8
7 - 5	-	-	-	14,811	376.8	55.04	11,643	41.8	51.8

Table 3: Results for the *mastermind* game. We use *size* to denote the representation size (averaged over all time steps), $T_{offline}$ for runtimes of the offline phase and T_{inf} for the runtime to compute the forward message for 1 time step.

Domain	$T_{offline}$ (s)	0% evidence size #edges	0% evidence T_{inf} (s)	33% evidence size #edges	33% evidence T_{inf} (s)	66% evidence size #edges	66% evidence T_{inf} (s)
people		×1000		×1000		×1000	
3	0.01	0.9	0.01	0.5	0.01	0.4	0.01
4	0.01	39	0.01	14	0.01	3	0.01
5	0.01	45	0.01	15	0.01	4	0.01
6	0.02	330	0.07	27	0.04	13	0.02
7	0.2	2,541	0.6	210	0.5	40	0.15
8	0.8	21,889	40.1	1,281	5.9	84	0.73
9	12.1	-	-	4,170	75.6	423	16.4
10	11.0	-	-	-	-	654	19.1

Table 4: Results for *sickness* network.

Hence, we conclude that dynamic $T_{\mathcal{P}}$ compilation allows us to efficiently exploit evidence to further push the boundaries of exact inference in dynamic relational domains.

7. Related Work

Knowledge compilation and weighted model counting has shown to be very effective for inference in probabilistic logic programs (De Raedt *et al.*, 2007; Riguzzi, 2007; Riguzzi and Swift, 2011; Fierens *et al.*, 2015) as well as graphical models (Chavira and Darwiche, 2005; Darwiche, 2009; Choi *et al.*, 2013). Exact compilation of a propositional formula is computational expensive, however, and one often has to resort to approximate techniques. One way to do so is to first convert the program into a propositional formula, as done for exact compilation, but then only compile selected subformulas (Renkens *et al.*, 2014) or feed the formula to a sampling algorithm, e.g. MC-SAT (Poon and Domingos, 2006).

In case also construction of the complete propositional formula becomes infeasible, one can transform the original program to an approximate, simplified program that represents, ideally, a similar probability distribution (Renkens *et al.*, 2012). Other approaches employ forward reasoning to directly sample on the logic program, e.g., (Milch *et al.*, 2005; Goodman *et al.*, 2008; Gutmann *et al.*, 2011; Nitti *et al.*, 2014), but these do not provide guaranteed lower or upper bounds on the probability of the queries. Anytime PLP algorithms based on backward reasoning have been proposed in the past but they do not allow to answer multiple queries in parallel (Poole, 1993; De Raedt *et al.*, 2007). The problem of highly cyclic domains has recently also been addressed using lazy clause generation (Aziz *et al.*, 2015), but only for exact inference.

Probabilistic logic programs under the distribution semantics define a distribution over possible worlds, which randomly fixes the truth values of probabilistic facts and then permits any type of logical reasoning within a possible world. While our approach focusses on stratified programs with finite support, the fixpoint operator is more recently also extended towards general normal programs with function symbols (Bogaerts and Van den Broeck, 2015; Riguzzi, 2016). A second class of probabilistic Prologs, including Stochastic Logic Programs (SLPs) (Muggleton, 1996), uses a different approach, where a distribution over the groundings of a query is defined based on a distribution over the derivations in the query's SLD tree, making an independent decision on which branch to take at every node. The semantics is thus closely tied to backward reasoning, and our forward reasoning based approach does not easily apply in this setting.

Dynamic programs in PLP, one of the use cases considered in this paper,

are typically handled by dedicated methods that are extensions of existing inference techniques. Currently, inference in dynamic relational domains relies on approximate techniques (de Salvo Braz *et al.*, 2008; Nitti *et al.*, 2013) or requires that each rule in the program considers a transition over time steps (Thon *et al.*, 2011). Other approaches focus on models that are acyclic or propositional (Murphy, 2002; Sato, 1995; Vlasselaer *et al.*, 2016). Online or dynamic inference has also been considered in the context of Markov logic networks, but this only with approximate inference, e.g. (Kersting *et al.*, 2009; Geier and Biundo, 2011).

8. 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. Furthermore, we have extended $T_{\mathcal{P}}$-compilation towards dynamic domains and have shown that it efficiently exploits given observations to scale-up inference.

Acknowledgments

We wish to thank Bart Bogaerts for useful discussions and Adnan Darwiche and Arthur Choi for support with the SDD package. Jonas Vlasselaer is supported by IWT (agency for Innovation by Science and Technology). Angelika Kimmig is supported by FWO (Research Foundation-Flanders).

References

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.

Bart Bogaerts and Guy Van den Broeck. Knowledge compilation of logic programs using approximation fixpoint theory. *Theory and Practice of Logic Programming*, 15(4-5):464–480, 2015.

X. Boyen and D. Koller. Tractable Inference for Complex Stochastic Processes. In *In Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence (UAI)*, 1998.

Mark Chavira and Adnan Darwiche. Compiling Bayesian Networks with Local Structure. In *In Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*, 2005.

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

Mark Chavira, Adnan Darwiche, and Manfred Jaeger. Compiling relational Bayesian networks for exact inference. *International Journal of Approximate Reasoning*, 42(1):4–20, 2006.

Arthur Choi, Doga Kisa, and Adnan Darwiche. Compiling Probabilistic Graphical Models using Sentential Decision Diagrams. In *Proceedings of the 12th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU)*, 2013.

Keith L Clark. Negation as failure. *Logic and databases*, pages 293–322, 1978.

Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of AI Research*, 17:229–264, 2002.

Adnan Darwiche. New Advances in Compiling CNF into Decomposable Negation Normal Form. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI)*, 2004.

Adnan Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009.

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

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

Luc De Raedt and Angelika Kimmig. Probabilistic (logic) programming concepts. *Machine Learning*, 100(1):5–47, 2015.

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.

L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton, editors. *Probabilistic Inductive Logic Programming — Theory and Applications*, volume 4911 of *Lecture Notes in Artificial Intelligence*. Springer, 2008.

Rodrigo de Salvo Braz, Nimar Arora, Erik Sudderth, and Stuart Russell. Open-universe state estimation with dblog. In *NIPS 2008 Workshop*, 2008.

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*, 15(03):358–401, May 2015.

Thomas Geier and Susanne Biundo. Approximate Online Inference for Dynamic Markov Logic Networks. In *Proceedings of the 23rd International Conference on Tools with Artificial Intelligence (ICTAI)*, 2011.

L. Getoor and B. Taskar, editors. *An Introduction to Statistical Relational Learning*. MIT Press, 2007.

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)*, 2008.

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.

Kristian Kersting, Babak Ahmadi, and Sriraam Natarajan. Counting Belief Propagation. In *Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence (UAI)*, 2009.

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 19th International Joint Conference on Artificial Intelligence (IJCAI)*, 2005.

S. H Muggleton. Stochastic logic programs. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, pages 254–264. IOS Press, 1996.

Kevin Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, UC Berkeley, Computer Science Division, July 2002.

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

Davide Nitti, Tinne De Laet, and Luc De Raedt. A particle filter for hybrid relational domains. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2013.

Davide Nitti, Tinne De Laet, and Luc De Raedt. Relational object tracking and learning. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2014.

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.

Avi Pfeffer. *Practical Probabilistic Programming*. Manning Publications, 2014.

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

Hoifung Poon and Pedro Domingos. Sound and Efficient Inference with Probabilistic and Deterministic Dependencies. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI)*, 2006.

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.

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)*, 2014.

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.

Fabrizio Riguzzi. A Top Down Interpreter for LPAD and CP-Logic. In *Proceedings of the 10th Congress of the Italian Association for Artificial Intelligence (AI*IA)*, 2007.

Fabrizio Riguzzi. The Distribution Semantics for Normal Programs with Function Symbols. *International Journal of Approximate Reasoning (Special Issue on Probabilistic Logic Programming)*, (Accepted), 2016.

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

Dan Suciu, Dan Olteanu, R. Christopher, and Christoph Koch. *Probabilistic Databases*. Morgan & Claypool Publishers, 1st edition, 2011.

Ingo Thon, Landwehr Niels, and Luc De Raedt. Stochastic relational processes: Efficient inference and applications. *Machine Learning*, 82(2):239–272, February 2011.

Guy Van den Broeck and Adnan Darwiche. On the Role of Canonicity in Knowledge Compilation. In *Proceedings of the 29th Conference on Artificial Intelligence (AAAI)*, 2015.

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.

Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The Well-founded Semantics for General Logic Programs. *J. ACM*, 38(3):619–649, July 1991.

Joost Vennekens, Sofie Verbaeten, and Maurice Bruynooghe. Logic programs with annotated disjunctions. In *Proceedings of the 20th International Conference on Logic Programming (ICLP)*, 2004.

Joost Vennekens, Marc Denecker, and Maurice Bruynooghe. CP-logic: A language of causal probabilistic events and its relation to logic programming. *Theory and Practice of Logic Programming*, 9(3), 2009.

Jonas Vlasselaer, Guy Van den Broeck, Angelika Kimmig, Wannes Meert, and Luc De Raedt. Anytime inference in probabilistic logic programs with Tp-compilation. In *Proceedings of 24th International Joint Conference on Artificial Intelligence (IJCAI)*, July 2015.

Jonas Vlasselaer, Wannes Meert, Guy Van den Broeck, and Luc De Raedt. Exploiting local and repeated structure in dynamic Bayesian networks. *Artificial Intelligence*, 232:43–53, March 2016.