Implementation and Performance of Probabilistic Inference Pipelines

Dimitar Shterionov(✉) and Gerda Janssens

Department of Computer Science, KU Leuven,
Celestijnenlaan 200A, bus 2402, 3001 Heverlee, Belgium
{dimitar.shterionov,gerda.janssens}@cs.kuleuven.be

Abstract. In order to handle real-world problems, state-of-the-art probabilistic logic and learning frameworks, such as ProbLog, reduce the expensive inference to an efficient Weighted Model Counting. To do so ProbLog employs a sequence of transformation steps, called an *inference pipeline*. Each step in the probabilistic inference pipeline is called a *pipeline component*. The choice of the mechanism to implement a component can be crucial to the performance of the system. In this paper we describe in detail different ProbLog pipelines. Then we perform a empirical analysis to determine which components have a crucial impact on the efficiency. Our results show that the Boolean formula conversion is the crucial component in an inference pipeline. Our main contributions are the thorough analysis of ProbLog inference pipelines and the introduction of new pipelines, one of which performs very well on our benchmarks.

1 Introduction

Probabilistic Logic and Learning (PLL) software such as ProbLog [7,12] provides a machinery to derive new knowledge from uncertain data. Performing probabilistic inference or learning efficiently is a challenging task. In order to handle real-world problems state-of-the-art PLL frameworks employ knowledge compilation that reduces the initial inference or learning task into a weighted model counting (WMC) problem. Knowledge compilation converts a Boolean formula into another formula with special properties. These properties allow efficient weighted model counting on the compiled formula.

The inference mechanism of ProbLog encompasses a sequence of transformation steps in order to first compile the initial ProbLog program together with a set of query and evidence atoms and second to perform WMC on the compiled form. We call this transformation sequence an *inference pipeline* and the transformation steps – *pipeline components*. There are four components in a ProbLog pipeline – *Grounding*, *Boolean formula conversion*, *Knowledge compilation* and *Evaluation*. Each of them can be implemented with a different tool or algorithm, as long as the input/output requirements are respected. For example, ProbLog1 [7] uses knowledge compilation to ROBDDs while ProbLog2 [8] uses knowledge compilation to sd-DNNFs. In order to comply with these requirements

E. Pontelli and T.C. Son (Eds.): PADL 2015, LNCS 9131, pp. 90–104, 2015.
DOI:10.1007/978-3-319-19686-2_7

it may be the case that an intermediate data formatting is needed. For example, the Boolean formula that needs to be compiled to ROBDD or sd-DNNF needs to be formatted as a BDD script or a CNF accordingly.

The performance of ProbLog pipelines depends on (i) how components are implemented, i.e., what tools or algorithms are used in order to convey the necessary transformations; and (ii) how they are linked together, i.e., how the output from one component is used as input for the next one. In this paper we investigate different implementations of each component in order to not only determine the optimal pipelines but also the components with crucial impact on the overall performance.

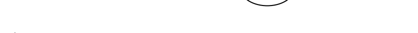
We compose 14 inference pipelines by substituting one algorithm by another for the same component when this is feasible. Then we evaluate their performance on 7 benchmark sets in order to determine the crucial component(s). These benchmarks can be considered as standard ProbLog benchmarks as they have been used in previous research to test different aspects of ProbLog inference and they cover different kinds of ProbLog programs. Our contribution is twofold – on the one hand it is the extensive analysis of ProbLog inference pipelines; and on the other the introduction of new inference pipelines, one of which performs very well on our benchmarks.

The paper is structured as follows. Section 2 gives background information on the ProbLog language as well as on weighted model counting for ProbLog inference. Section 3 presents our analysis of the different components. In Section 4 we present our experiments and discuss the results. Section 5 concludes our paper and discusses some possibilities for future research.

2 Background

2.1 The Probabilistic Logic and Learning Language ProbLog

ProbLog [7,12] is a general purpose Probabilistic Logic and Learning (PLL) programming language. It extends Prolog with probabilistic facts which encode uncertain knowledge. Probabilistic facts have the form $p_i :: f_i$, where p_i is the probability label of the fact f_i. Prolog rules define the logic consequences of the probabilistic facts. Fig. 1 shows a probabilistic graph and its encoding as a ProbLog program. The fact `0.6::e(a, b).` expresses that the edge between nodes `a` and `b` exists with probability 0.6.



a) A probabilistic graph.

```
0.6::e(a, b).  0.3::e(a, c).  0.8::e(b, c).
0.4::e(b, d).  0.7::e(c, d).
p(X, Y):- e(X, Y).
p(X, Y):- e(X, X1), p(X1, Y).
```

b) A ProbLog program.

Fig. 1. A probabilistic graph and its encoding as a ProbLog program. The `p/2` predicate defines the ("path") relation between two nodes: a path exists, if two nodes are connected by an edge or via a path to an intermediate node.

An atom which unifies with a probabilistic fact, called a *probabilistic atom* can be either true with the probability of the corresponding fact or false with (1−the probability). The choices of the truth values of all probabilistic atoms define a unique model of the ProbLog program called a *possible world*.

Let $\Omega = \{\omega_1, .., \omega_N\}$ be the set of possible worlds of a ProbLog program. Given that only probabilistic atoms have probabilities we see a single possible world ω_i as the tuple (ω_i^+, ω_i^-), where ω_i^+ is the set of probabilistic atoms in ω_i which are true and ω_i^- the set of probabilistic atoms which are false[1]. Probabilistic atoms are seen as independent random variables. A ProbLog program defines a distribution over possible worlds as given in Equation 1 where p_i denotes the probability of the atom a_i.

$$P(\omega_i) = \prod_{a_j \in \omega_i^+} p_j \prod_{a_j \in \omega_i^-} (1 - p_j) \tag{1}$$

A query q is true in a subset of the possible worlds: $\Omega^q \subseteq \Omega$. Each $\omega_i^q \in \Omega^q$ has a corresponding probability, computed by Equation 1. The (success or *marginal*) probability of q is the sum of the probabilities of all worlds in which q is true:

$$P(q) = \sum_{\omega_i \in \Omega^q} P(\omega_i) \tag{2}$$

Example 1. The query `p(a, d)` for the program in Fig. 1 is true if there is at least one path between nodes `a` and `d`. This holds in 15 out of the $2^4 = 32$ possible worlds each of them associated with a probability. Using Equation 2 gives the marginal probability $P($`p(a, d)`$) = 0.54072$.

The task of computing the marginal probability of a query (i.e. the *MARG* task) is the most basic inference task of ProbLog. ProbLog can also compute the conditional probability of the query given evidence (the *COND* task).

Example 2. For the program in Fig.1, the query `p(a,d).` and evidence `e(a,b)`$= false$ ProbLog computes the conditional probability $P($`p(a,d)`$|$`e(a,b)`$= false) = 0.21$.

2.2 Weighted Model Counting by Knowledge Compilation

Enumerating the possible worlds of a ProbLog program and computing the (marginal) probability of a query according to Equation 2 is a straightforward approach for probabilistic inference. Because the number of possible worlds grows exponentially with the increase of the number of probabilistic facts in a ProbLog program, this approach is considered impractical.

[1] The union $\omega_i^+ \cup \omega_i^-$ is the set of all possible ground probabilistic atoms of the ProbLog program with the truth value assignments specific for the possible world ω_i; the intersection $\omega_i^+ \cap \omega_i^-$ is the empty set.

In order to avoid the expensive enumeration of possible worlds the inference mechanism of ProbLog uses knowledge compilation and an efficient weighted model counting method. Model Counting is the process of determining the number of models of a formula φ. The *Weighted* Model Count (WMC) of a formula φ is the sum of the weights that are associated with each model of φ. For a given ProbLog program L with a set of possible worlds Ω the WMC of a formula φ coincides with Equation 2 when there is a bijection between the models (and their weights) of φ and the possible worlds (and their probabilities) in Ω.

The task of Model Counting (and also its specialization Weighted Model Counting) is in general a $\#P$-complete problem. Its importance in SAT and in the Statistical Relational Learning and Probabilistic Logic and Learning communities has lead to the development of efficient algorithms [5] which have found their place in ProbLog. By using knowledge compilation the actual WMC can be computed linearly to the size of the compiled (arithmetic) circuit [5, Chapter12].

3 Inference Pipeline

In order to transform a ProbLog inference task into a WMC problem ProbLog uses a sequence of transformation steps, called an *inference pipeline*. The starting point of the inference pipeline is a ProbLog program together with a (possibly empty) set of query and evidence atoms. The four main transformation steps, i.e. *components* that compose an inference pipeline are: *Grounding*, *Boolean formula conversion*, *Knowledge compilation* and *Evaluation*. The grounding generates a propositional instance of the input ProbLog program. It ignores the probabilistic information of that program, i.e. the probability label of each probabilistic fact. Second, the propositional instance is converted to a Boolean formula. The Boolean formula and the propositional instance have the same models. Third, the Boolean formula is compiled into a *negation normal form* (NNF) with certain properties which allow efficient model counting. Finally, this NNF is converted to an arithmetic circuit which is associated with the probabilities of the input program and weighted model counting is performed.

Each component can be implemented by different tools or algorithms, as long as the input/output requirements between components are respected. For example, ProbLog1 [7] uses knowledge compilation to *Reduced Ordered Binary Decision Diagrams* (ROBDDs) [1] in order to reduce the inference task to a tractable problem. Later, [9] illustrates an approach for ProbLog inference by compilation to a *smooth, deterministic, Decomposable Negation Normal Form* (sd-DNNF) [6]. Fig. 2 gives an overview of the different approaches that can be used to implement a component and how they can be linked to form an inference pipeline. In the remaining of this section we present in detail each pipeline component and the underlying algorithms used to accomplish the necessary transformations.

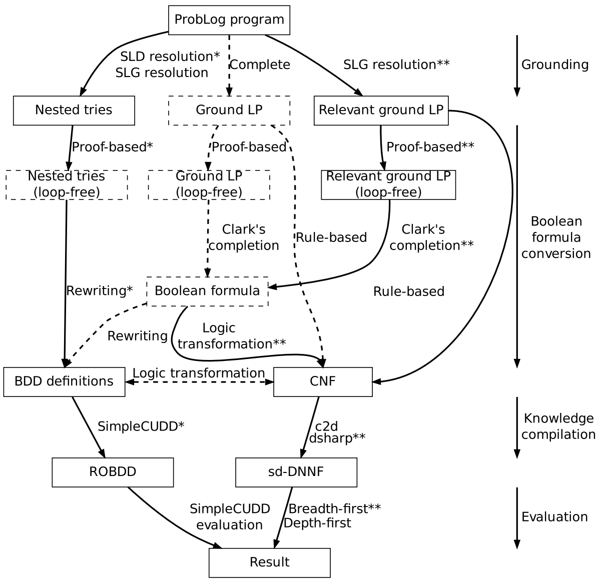


Fig. 2. ProbLog pipelines. Nodes represent Input/output formats. Each edge states a transformation and points from the output to the input. Solid edges define an existing pipeline. Default pipelines are indicated by (*) for MetaProbLog/ProbLog1 and (**) for ProbLog2. Dashed edges indicate a nonexistent pipeline. Dashed nodes indicate intermediate data formats. The input ProbLog program may contain query and evidence atoms. Vertical arrows alongside the graph indicate the components.

3.1 Grounding

A naive grounding approach is to generate the *complete* set of possible instances of the initial ProbLog program according to the values a variable can be bound to. Such a complete grounding may result in extremely big ground programs. It is more efficient with respect to the size of the grounding and the time for its generation to focus on the part of the ProbLog program which is relevant to an atom of interest. A ground ProbLog program is relevant to an atom q if it contains only relevant atoms and rules. An atom is relevant if it appears in some *proof* of q. A ground rule is relevant with respect to q if its head is a relevant atom and its body consists of relevant atoms. It is safe to confine to the ground program relevant to q because the models of the relevant ground program are the same as the models of the initial ProbLog program that entail the atom q. That is, the relevant ground program captures the distribution $P(q)$ entirely (proof of correctness can be found in [8], Theorem 1).

To determine the relevant grounding a natural mechanism is SLD resolution. Each successful SLD derivation for a query q determines one proof of q – a conjunction of ground literals. Naturally, all proofs to a query form a disjunction and therefore, can be represented as a Boolean formula in DNF. An SLD derivation may be infinite, e.g., in case of cyclic programs. In order to detect cycles (i) auxiliary code can be introduced to the input ProbLog program in order to store and compare intermediate results or (ii) SLG resolution [2] (that is, SLD with tabling) can be used instead. Adding auxiliary code as in (i) can slow down inference and is susceptible to user errors. That is why (ii), i.e. SLG resolution, is preferable for ProbLog inference.

We distinguish between two representations of the relevant grounding of a ProbLog program. ProbLog1 uses the **nested trie** structure as an intermediate representation of the collected proofs. If SLD resolution is used (that is, no tabling is invoked)[2] there is only one trie. ProbLog2 considers the **relevant ground logic program** with respect to a set of query and evidence atoms.

3.2 Boolean Formula Conversion

Logic Programs (LP) use the Closed World Assumption (CWA), which basically states that if an atom cannot be proven to be true, it is false. In contrast, First-Order logic (FOL) has different semantics: it does not rely on the CWA. Consider the (FOL) theory $\{q \leftarrow p\}$ which has three models: $\{\neg q, \neg p\}$, $\{q, \neg p\}$ and $\{q, p\}$. Its syntacticly equivalent LP (`q :- p.`) has only one model, namely $\{\neg q, \neg p\}$. In order to generate a Boolean formula from nested tries (ProbLog1, MetaProbLog) or a relevant ground LP (ProbLog2), it is required to make the transition from LP semantics to FOL semantics. When the grounding does not contain cycles it suffices to take the Clark's completion of that program [10,11]. When the grounding contains cycles it is proven that the Clark's completion does not result in an equivalent Boolean formula [11]. To handle cyclic groundings ProbLog employs one of two methods. The **proof-based** approach [14] basically removes proofs containing cycles as they do not contribute to the probability. This approach is query-directed, i.e. it considers a set of queries and traverses their proofs. The **rule-based** approach is inherited from the field of Answer Set Programming. It rewrites a rule with cycles to an equivalent rule and introduces additional variables in order to disallow cycles [11].

Once the cycles are handled, ProbLog1 rewrites the Boolean formula encoded in the nested tries as **BDD definitions**. A BDD definition [14] is a formula with a head and a body, linked with equivalence. The body of a BDD definition contains literals and/or heads of other BDD definitions combined by conjunctions or disjunctions. The logic operators are translated to arithmetic functions. A BDD script is a set of BDD definitions.

In the case of ProbLog2, the Clark's completion of the loop-free relevant ground LP is used to generate a Boolean formula. This Boolean formula is then rewritten in **CNF**. It can also be rewritten to **BDD definitions**. It is important to exploit the structure of this Boolean formula during the rewrite, otherwise the BDD script may blow up in size.

Example 3. For the ProbLog program in Fig 1 b) and the query `p(b, d)` the Boolean formula associated with the completion of the relevant ground LP is: $(p_{bd} \iff (e_{bd} \vee (e_{bc} \wedge p_{cd}))) \wedge (p_{cd} \iff e_{cd})$, where p_{xy} and e_{xy} denote `p(x, y)` and `e(x, y)` respectively. Following are its equivalent representations as a CNF and BDD definitions where $a0$ stands for an auxiliary Boolean variable:

CNF:	$(\neg p_{bd} \vee e_{bd} \vee a0) \wedge (p_{bd} \vee \neg e_{bd}) \wedge (p_{bd} \vee \neg a0) \wedge (a0 \vee \neg e_{bc} \vee \neg p_{cd}) \wedge$ $(\neg a0 \vee e_{bc}) \wedge (\neg a0 \vee p_{cd}) \wedge (p_{cd} \vee \neg e_{cd}) \wedge (\neg p_{cd} \vee e_{cd})$
BDD definitions:	p_{bd} `=` e_{bd} `+ a0` `a0 =` e_{bc} `*` p_{cd} p_{cd} `=` e_{cd}

[2] ProbLog1 allows the user to select whether to use tabling or not. ProbLog2 always uses tabling.

Example 4. A CNF can be rewritten as BDD definitions and vice-versa by a set of logical transformations. The following BDD definitions are generated from the CNF in Example 3 and are equivalent to the formula in Example 3:

BDD definitions:	a1 = p_{bd} + e_{bd} + a0	a2 = p_{bd} + ~e_{bd}	a3 = p_{bd} + ~a0	a4 = a0 + ~e_{bc} + ~p_{cd}
	a5 = a0 + e_{bd}	a6 = ~a0 + p_{cd}	a7 = p_{cd} + ~e_{cd}	a8 = ~p_{cd} + e_{cd}
	a9 = a1 * a2 * a3 * a4 * a5 * a6 * a7 * a8			

Example 3 shows how a Boolean formula that originates from Clark's completion of the relevant ground LP can easily be rewritten in CNF as well as in BDD definitions. It also shows that a CNF representation of such a formula is less succinct ([6]) than the representation as BDD definitions. If though a CNF formula is converted to BDD definitions as in Example 4 the BDD script blows up in size. For the overall performance of a pipeline it is crucial to avoid such a transformation. This phenomenon is discussed among others in [17]. In [8,9] the authors consider a ProbLog pipeline in which a CNF formula is transformed into BDD definitions as shown in Example 4, i.e. a relevant ground LP is first converted to a Boolean formula in CNF which subsequently is converted to a BDD script. Their experiments confirm that such an approach is inefficient for ProbLog inference. We do not consider further inference pipelines which include a transformation from CNF to BDD definitions. To the contrary, we introduce a *new pipeline* which transforms the relevant ground program directly into BDD definitions avoiding the blow up of the BDD script (see Table 1, pipeline $P4$).

3.3 Knowledge Compilation and Evaluation

ProbLog uses knowledge compilation to compile the Boolean formula to a negation normal form (NNF) that has the properties *determinism*, *decomposability* and *smoothness* [6]. Such an NNF is then used for efficient WMC. In ProbLog's inference pipelines two target compilation languages have been exploited so far: (i) **ROBDDs** [1] common for ProbLog1 (and MetaProbLog [13, Chapter6]) and (ii) **sd-DNNFs** [6] employed by ProbLog2.

To compile a Boolean formula to a ROBDD ProbLog implementations use **SimpleCUDD** (www.cs.kuleuven.be/~theo/tools/simplecudd.html). Compiling to sd-DNNF is done with the **c2d** [3,4] or **dsharp** [15] compilers.

After the knowledge compilation step, the compiled formula is traversed in order to compute the probabilities (i.e. the WMC) for the given query(ies) – the evaluation step. ProbLog employs two approaches to traverse sd-DNNFs: **breadth-first** and **depth-first**[3]) and one to traverse ROBDDs.

Sections 3.1 to 3.3 describe the components of the two mainstream ProbLog pipelines – ProbLog1 and ProbLog2. The subprocesses which are used in these pipelines constitute a set of interchangeable components which may form other working pipelines. Fig. 2 gives an overview of the possible ProbLog pipelines. The

[3] To invoke one of these two options in ProbLog2 one specifies either the *fileoptimized* (default) for the breadth-first implementation or *python* for the depth-first implementation as evaluation options.

link between different components depends on the compatibility of the output of a preceding subprocess with the input requirements of the next one. For example, c2d cannot compile BDD definitions but requires CNFs. Earlier it was shown that some pipelines are certain to perform worse than others: pipelines with (naive) complete grounding; pipelines in which a CNF is converted to BDD definitions (cf. Section 3.2). In addition, we prefer using SLG resolution for grounding instead of SLD resolution in order to avoid possible cycles. This leaves the 14 pipelines shown in Table 1. $P4$ and $P9..P12$ are previously unexploited pipelines for ProbLog inference.

Table 1. Pipelines used in the experiments. $X \rightarrow Y$ stands for a transformation X and the output representation Y (see Fig. 2).

	Grounding	Boolean formula conversion	Knowledge compilation	Evaluation	New Pipeline
$P0$	SLG→Rel. gr. LP	Proof-based→CNF	c2d→sd-DNNF	Breadth-first	No
$P1$	SLG→Rel. gr. LP	Proof-based→CNF	c2d→sd-DNNF	Depth-first	No
$P2$	SLG→Rel. gr. LP	Proof-based→CNF	dsharp→sd-DNNF	Breadth-first	No
$P3$	SLG→Rel. gr. LP	Proof-based→CNF	dsharp→sd-DNNF	Depth-first	No
$P4$	SLG→Rel. gr. LP	Proof-based→BDD def.	SimpleCUDD→ROBDD	SimpleCUDD	**Yes**
$P5$	SLG→Rel. gr. LP	Rule-based→CNF	c2d→sd-DNNF	Breadth-first	No
$P6$	SLG→Rel. gr. LP	Rule-based→CNF	c2d→sd-DNNF	Depth-first	No
$P7$	SLG→Rel. gr. LP	Rule-based→CNF	dsharp→sd-DNNF	Breadth-first	No
$P8$	SLG→Rel. gr. LP	Rule-based→CNF	dsharp→sd-DNNF	Depth-first	No
$P9$	SLG→Nested tries	Proof-based→CNF	c2d→sd-DNNF	Breadth-first	**Yes**
$P10$	SLG→Nested tries	Proof-based→CNF	c2d→sd-DNNF	Depth-first	**Yes**
$P11$	SLG→Nested tries	Proof-based→CNF	dsharp→sd-DNNF	Breadth-first	**Yes**
$P12$	SLG→Nested tries	Proof-based→CNF	dsharp→sd-DNNF	Depth-first	**Yes**
$P13$	SLG→Nested tries	Proof-based→BDD def.	SimpleCUDD→ROBDD	SimpleCUDD	No

4 Evaluation

4.1 Experimental Set-Up

Our experiments aim to determine the impact of the different components on the performance of the 14 pipelines. And more specifically, the components which have a crucial impact on the overall performance.

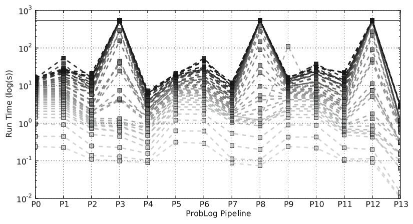
We run the 14 pipelines on 7 benchmark sets with in total 319 benchmark programs: "Alzheimer" [7], "Balls" [20], "Dictionary" [18], "Grid" [8], "Les Miserables" [18], "Smokers" [16], "WebKB" [9]. The programs from the "Alzheimer", "Dictionary", "Les Miserables" and "WebKB" are built from real-world data; the rest are based on artificial data.

The benchmark programs we use encode different directed probabilistic graphs. The graphs corresponding to the "Grid" benchmarks are acyclic with a hierarchical structure and maximum in/out degree of 3. The rest are cyclic; the ones in the "Les Miserables" and the "Dictionary" are sparse graphs (with density < 0.0012 and < 0.0002 respectively). Probabilistic graphs are encoded as shown in Fig. 1. The queries to these programs ask for the probability a path exists between two nodes.

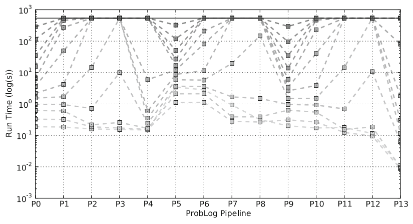
A program from the "Smokers" or "WebKB" benchmark sets contains multiple queries. The rest contain one query. The variety of these benchmarks ensures a close to realistic estimate of the general performance of ProbLog pipelines. The programs from the "Balls" benchmark set use annotated disjunctions [21] to encode random events with multiple outcomes. They are acyclic.

Our benchmarks have been used previously to evaluate different aspects of ProbLog implementations. The benchmarks from the "Alzheimer" set were used to motivate the development and test the performance of the first ProbLog system. The "Smokers" and "WebKB" benchmark sets are used for testing ProbLog2, i.e. different loop-breaking and knowledge compilation approaches. Also, the "Grid" benchmark set was developed in the context of ProbLog2 and to compare the knowledge compilation to sd-DNNFs with knowledge compilation to ROBDDs. The "Balls" benchmark set is used to test the performance of a new encoding of Annotated Disjunctions for ProbLog programs (mainly affecting the grounding). That is why we believe our experiments will allow to clearly determine the crucial components in the inference pipeline.

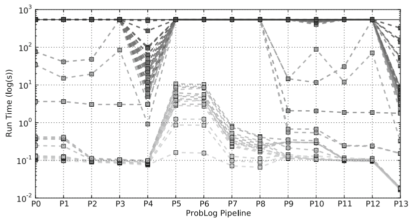
In our experiments, we measure the run times of each component while performing the MARG or the COND task for the given query(ies) and evidence. Because the sd-DNNF compilers are non-deterministic [3, 15], i.e. for the same CNF the compiled sd-DNNFs may differ, we run all tests 5 times and report the



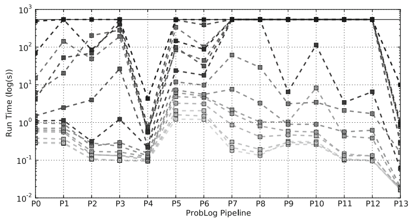
a) "Balls" benchmark set.



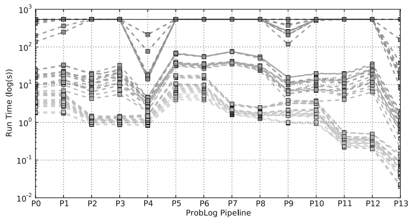
b) "Grid" benchmark set.



c) "Les Miserables" benchmark set.



d) "Smokers" benchmark set.



e) "WebKB" benchmark set.

Fig. 3. Run times for ProbLog pipelines performing MARG inference

average run time. Previous tests with these compilers within ProbLog have shown that the average time for 5 runs gives a realistic estimate on the performance. We set a time-out of 540 seconds for each run.

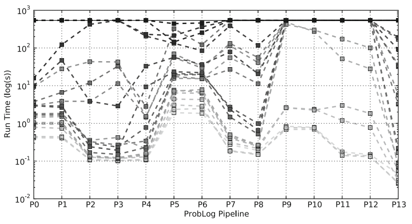
Section 4.2 presents our experimental results. A discussion follows in Section 4.3. Detailed description of our benchmarks, complete results and color diagrams can be found in [19]. Enlarged and color version of the diagrams in Fig. 3 and Fig. 4 are available in http://people.cs.kuleuven.be/~dimitar.shterionov/pipeline_diagrams.pdf. Our benchmarks can be found at http://people.cs.kuleuven.be/~dimitar.shterionov/benchmarks_pipelines.zip. In the future we would like to extend this set with new problems in order to improve generality of our conclusions.

4.2 Results

We present the total run time (the sum of the grounding, Boolean formula conversion, knowledge compilation and evaluation times) of each pipeline for a benchmark program executing MARG or COND inference. The reason to focus only on the total run time is that any change in the performance of two pipelines which share all but one component will be due to the different component. Whether the algorithm that implements the component, the compatibility with the input data or the output have an affect on the overall performance is not of importance. Rather, we are interested in how the different components' implementations influence the pipeline as a whole. To get an idea of the impact of individual components we compare the result for pipelines which differ by one component. For example, comparing pipelines $P0 - P8$ to pipelines $P9 - P13$ will determine the effect of the two different grounding approaches. Fig. 3 shows the total run time for performing MARG inference on the "Balls", "Grid", "Les Miserables", "Smokers" and "WebKB" benchmark sets. The results from the "Les Miserables" benchmarks are similar to the "Alzheimer" and the "Dictionary"; although the results from the "Smokers" benchmarks are similar to the "WebKB" we show both diagrams so that later they can be compared to the results from performing COND inference shown in Fig. 4.

In each figure a horizontal line is associated with one benchmark program and shows the total run time (thus the lower the better) of each pipeline (x-axis) executing the MARG or the COND task on that program. We use a logarithmic scale for the time axis (the y-axis). We present the lines in different shades of gray relative to the size of the dependency graph representing the program. The black line parallel to the x-axis indicates the 540^{th} second, that is, the time-out.

We also give the number of timeouts that occurred for each pipeline performing MARG and COND inference in Table 2 and Table 3 respectively. They show the total number of timeouts and the relative number of timeouts with respect to the total number of programs in a benchmark set for which at least one pipeline terminated successfully. For example, $P4$ times out for a total of 11 benchmarks when executing COND inference (see Table 3); 2 of the programs that time out are from the "Smokers" set and 9 from the "WebKB" set; in total 20 programs of the "Smokers" and 48 of the "WebKB" benchmark sets have been successfully executed; we compute the relative number of timeouts as $2/20 + 9/48 = 0.2875$.


a) “Smokers” benchmark set.

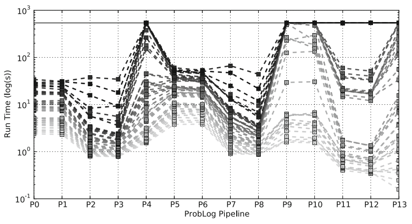

b) “WebKB” benchmark set.

Fig. 4. Run times for ProbLog pipelines performing COND inference

Table 2. Number of benchmark programs for which MARG inference times out

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
Total:	46	53	62	84	8	144	145	158	177	48	47	68	89	14
Total (relative):	3.14	3.48	4.35	5.34	0.72	7.76	7.94	8.8	9.28	3.95	4.12	5.04	5.71	1.64

Table 3. Number of benchmark programs for which COND inference times out

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
Total:	3	3	3	4	11	0	0	4	4	42	42	27	27	21
Total (relative):	0.15	0.15	0.15	0.2	0.29	0.0	0.0	0.2	0.2	1.25	1.25	0.94	0.94	0.55

4.3 Discussion

We discuss the results from our experiments with the MARG task separately from the COND task. This is because computing the conditional probabilities in MetaProbLog (whose components we use to build other pipelines) differs from how conditional probabilities are computed in ProbLog2. The difference is due to the way evidence is processed.

MARG Inference. *Grounding* Comparing pipelines $P0, .., P4$ to $P9, .., P13$ in Fig. 3 shows that grounding to relevant ground LP and grounding to nested tries have similar impacts on the performance. The default MetaProbLog pipeline, $P13$ and pipeline $P4$ differ on the grounding representation. $P13$ appears to be faster than the rest in almost all of the cases. The timeouts in Table 2 though show that pipelines which use the relevant ground LP representation can solve (relatively) more problems than the ones using the nested tries. In particular, we notice that $P4$ outperforms $P13$. The effect of the one grounding representation compared to the other is though small therefore we can state that the choice of grounding representation is not crucial for the total inference performance.

Boolean Formula Conversion. When comparing pipelines $P0, .., P3$, to $P5, .., P8$ in Fig. 3 we observe that the Boolean formula conversion has a strong impact on the performance. By itself the time for conversion is not significant but it is the output Boolean formula that strongly influences the next components in the inference pipeline – knowledge compilation and evaluation. Knowledge compilation is computationally the most expensive task. The proof-based approach generates Boolean formulae which are easier to compile, i.e. the compilation time is lower than for the rule-based approach [19]. The time out results in Table 2 show that pipelines using the proof-based conversion time out 42% to 59%[4] less than pipelines using the rule-based approach.

For the effectiveness of the conversion of great importance is the presence of cycles in the grounding. We notice (Fig. 3 a) and b)) that pipelines using the rule-based conversion handle the acyclic graphs from the "Balls" and the "Grid" benchmark sets equally well or even better than some of the pipelines using the proof-based conversion. This is because the conversion does not need to handle any cycles and the rule-based conversion which simply traverses the ground program is not only faster (see Fig. 3 a) and Fig. 3 b)) but also generates easy-to-compile Boolean formulae.

These results show that the Boolean formula conversion is crucial for the inference pipeline.

Knowledge Compilation and Evaluation Knowledge compilation has the highest impact on the inference run time. Generally, knowledge compilation to ROBDDs is preferable for MARG inference (compare $P4$ and $P13$ to the rest in Fig. 3).

In the case of knowledge compilation to sd-DNNFs a pipeline which uses c2d shows better scalability compared to one with dsharp but is slower for the less complex problems. Furthermore, the breadth-first evaluation approach is in general preferable to the depth-first approach (compare $P0$ to $P1$ or $P11$ to $P12$ in Fig. 3 c)), although for the "Balls" benchmarks this evaluation approach performs poorly (see $P3$, $P8$ and $P12$ in Fig. 3 a)). The reason is the structure of the graph associated with the relevant ground LP – low out degree, i.e. 9, long paths from the root to the nodes.

COND Inference. The conditional probability of a query q given evidence $E = e$ is computed as the ratio $P(q|E = e) = \frac{P(q \wedge E=e)}{P(E=e)}$. First both the nominator and denominator need to be computed separately. Then their division gives the final result. MetaProbLog and ProbLog2 use different approaches when it comes to computing the conditional probabilities. In particular, there are differences regarding the grounding to nested tries and compiling to ROBDDs compared to grounding to a relevant ground LP and knowledge compilation to s-DDNNFs.

Grounding We notice from Fig. 4 a) and b) and Table 3 that grounding to nested tries has a negative effect on the overall performance as compared to grounding to a relevant ground LP. The former approach is: (i) for a query q and evidence $E = e$ a new query $q^{E=e}$ (i.e., $q \wedge E = e$) is created; (ii) $q^{E=e}$ and the atoms in E are proven in order to determine the relevant grounding (stored as nested tries). In the latter case, a query q and the atoms in E are used separately and not in a conjunction to determine the relevant ground LP. Although the two approaches result in very similar groundings, the evidence atoms and their predetermined values make a difference for the performance of the next components.

[4] We use the relative number of timeouts rather than the total number of timeouts in order to determine a more general interval.

Boolean Formula Conversion The Boolean formula is built by using either the proof-based or the rule-based method. In the case of pipelines $P0$ to $P9$ the Boolean formula (either represented as a CNF or as a BDD script) is augmented with clauses to state the truth values for the evidence atoms. They often help the knowledge compilation as they may prune parts of the compiled circuit. The positive effect is obvious when comparing pipelines $P0$ to $P4$ with $P9$ to $P13$ in Fig. 4 but also from Table 3.

Knowledge Compilation and Evaluation The additional clauses for the evidence added to the Boolean formula improve the performance for pipelines $P0$ to $P3$ and $P5$ to $P9$ as compared to executing the MARG task. The two pipelines using ROBDDs ($P4$ and $P13$) do not perform well. A reason for the decreased performance of these pipelines is that for multiple queries (including evidence) it is required to build and evaluate a forest of ROBDDs. In order to compute the conditional probability of a query q given evidence $E = e$ a ROBDD for the conjunction $q \wedge E = e$ is added to the ROBDD forest even when the conjunction is false ($P(q \wedge E = e) = 0.0$ therefore $P(q|E = e) = 0.0$), thus performing unnecessary operations. Indeed, this slow down is observed for the "WebKB" benchmark programs where a lot of the queries are false given that the evidence is true. Fig. 4 shows that the ROBDD-based pipelines ($P4$ and $P13$) do not scale as well as in the case of MARG inference. Which is also confirmed by Table 3.

5 Conclusions and Future Work

In this paper we presented a detailed description of the inference pipelines of ProbLog and analyzed their performance on 7 benchmark sets. Our analysis shows that the Boolean formula conversion has a crucial impact on the performance of the inference pipeline for both MARG and COND tasks. We showed that in most of the cases pipelines which use a *proof-based conversion*, *knowledge compilation to sd-DNNF with c2d* and the *breadth-first evaluation* approach and pipelines which use *proof-based conversion* and *compilation to ROBDDs* perform better than the rest. $P4$ and $P13$ are the most efficient pipelines for our benchmarks on performing MARG inference. $P13$ is the default pipeline of MetaProbLog. $P4$ is one of the new pipelines we introduce with this paper (combining ProbLog2 with ROBDDs).

We also showed that for COND inference it is crucial how the evidence is handled. Pipelines which use *compilation to sd-DNNF* and *breadth-first evaluation* outperform the rest. The most efficient pipeline for computing the COND task is $P0$. We also determined that this difference is due to how evidence is handled.

Our analysis determines two main directions for future research: (i) to improve the Boolean formula conversion component and (ii) to investigate how to improve ROBDDs with respect to computing conditional probabilities. Furthermore, pipeline $P4$ which combines the grounding of ProbLog2 with the knowledge compilation and evaluation of MetaProbLog via a direct conversion of the (loop-free) relevant ground LP to BDD definitions shows very promising results. To determine its actual place among the different ProbLog implementations we plan to further evaluate its performance on all inference and learning tasks supported by ProbLog.

References

1. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986)
2. Chen, W., Swift, T., Warren, D.S.: Efficient top-down computation of queries under the well-founded semantics. J. Log. Program. **24**(3), 161–199 (1995)
3. Darwiche, A.: A compiler for deterministic, decomposable negation normal form. In: Dechter, R., Sutton, R.S. (eds). AAAI/IAAI, pp. 627–634. AAAI Press/MIT Press (2002)
4. Darwiche, A.: New advances in compiling CNF into decomposable negation normal form. In: Proceedings of the 16th European Conference on Artificial Intelligence, pp. 328–332 (2004)
5. Darwiche, A.: Modeling and Reasoning with Bayesian Networks. Cambridge University Press (2009) (chapter 12)
6. Darwiche, A., Marquis, P.: A knowledge compilation map. Journal of Artificial Intelligence Research **17**, 229–264 (2002)
7. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: a probabilistic prolog and its application in link discovery. In Proceedings of the 20th International Joint Conference on Artificial Intelligence, pp. 2468–2473. AAAI Press (2007)
8. Fierens, D., Van Den Broek, G., Renkens, J., Shterionov, D., Gutmann, B., Thon, I., Janssens, G., de Raedt, L.: Inference and learning in probabilistic logic programs using weighted boolean formulas. Theory and Practice of Logic Programming, Special Issue on Probability, Logic and Learning 15(3), 358–401 (2015)
9. Fierens, D., Van den Broeck, G., Thon, I., Gutmann, B., De Raedt, L.: Inference in probabilistic logic programs using weighted CNF's. In: Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence, pp. 211–220 (2011)
10. Gutmann, B., Thon, I., De Raedt, L.: Learning the parameters of probabilistic logic programs from interpretations. In: Gunopulos, D., Hofmann, T., Malerba, D., Vazirgiannis, M. (eds.) ECML PKDD 2011, Part I. LNCS, vol. 6911, pp. 581–596. Springer, Heidelberg (2011)
11. Janhunen, T.: Representing normal programs with clauses. In: Proc. of the 16th European Conference on Artificial Intelligence, pp. 358–362. IOS Press (2004)
12. Kimmig, A., Demoen, B., De Raedt, L., Costa, V.S., Rocha, R.: On the implementation of the probabilistic logic programming language ProbLog. Theory and Practice of Logic Programming **11**, 235–262 (2011)
13. Mantadelis, T.: Efficient Algorithms for Prolog Based Probabilistic Logic Programming. PhD thesis, Informatics Section, Department of Computer Science, Faculty of Engineering Science, November 2012. Janssens, Gerda (supervisor)
14. Mantadelis, T., Janssens, G.: Dedicated tabling for a probabilistic setting. In: Hermenegildo, M.V., Schaub, T. (eds) ICLP (Technical Communications), vol. 7 of LIPIcs, pp. 124–133. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2010)
15. Muise, C., McIlraith, S.A., Beck, J.C., Hsu, E.I.: DSHARP: fast d-DNNF compilation with sharpSAT. In: Kosseim, L., Inkpen, D. (eds.) Canadian AI 2012. LNCS, vol. 7310, pp. 356–361. Springer, Heidelberg (2012)

16. De Raedt, L., Kersting, K.: Probabilistic inductive logic programming. In: De Raedt, L., Frasconi, P., Kersting, K., Muggleton, S.H. (eds.) Probabilistic Inductive Logic Programming. LNCS (LNAI), vol. 4911, pp. 1–27. Springer, Heidelberg (2008)
17. Rauzy, A., Châtelet, E., Dutuit, Y., Bérenguer, C.: A practical comparison of methods to assess sum-of-products. Rel. Eng. & Sys. Safety **79**(1), 33–42 (2003)
18. Shterionov, D., Janssens, G.: Data acquisition and modeling for learning and reasoning in probabilistic logic environment. In: Antunes, L., Pinto, H.S., Prada, R., Trigo, P. (eds) Proceedings of the 15th Portuguese Conference on Artificial Intelligence, pp. 298–312 (2011)
19. Shterionov, D., Janssens, G.: Crucial components in probabilistic inference pipelines: Data and results. Technical report, KU Leuven, 2014. Ref. number CW679. http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW679.pdf
20. Shterionov, D., Renkens, J., Vlasselaer, J., Kimmig, A., Meert, W., Janssens, G.: The most probable explanation for probabilistic logic programs with annotated disjunctions. In: Proceedings of the 24th International Conference on Inductive Logic Programming
21. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic programs with annotated disjunctions. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3132, pp. 431–445. Springer, Heidelberg (2004)