Compacting Boolean Formulae for Inference in Probabilistic Logic Programming

Theofrastos Mantadelis[1](✉), Dimitar Shterionov[2], and Gerda Janssens[2]

[1] CRACS & INESC TEC, Faculty of Sciences, University of Porto, Rua do Campo Alegre 1021/1055, 4169-007 Porto, Portugal
theo.mantadelis@dcc.fc.up.pt

[2] Department of Computer Science, KU Leuven, Celestijnenlaan 200A, 2402 3001 Heverlee, Belgium
{Dimitar.Shterionov,Gerda.Janssens}@cs.kuleuven.be

Abstract. Knowledge compilation converts Boolean formulae for which some inference tasks are computationally expensive into a representation where the same tasks are tractable. ProbLog is a state-of-the-art Probabilistic Logic Programming system that uses knowledge compilation to reduce the expensive probabilistic inference to an efficient weighted model counting. Motivated to improve ProbLog's performance we present an approach that optimizes Boolean formulae in order to speed-up knowledge compilation. We identify 7 Boolean subformulae patterns that can be used to re-write Boolean formulae. We implemented an algorithm with polynomial complexity which detects and compacts 6 of these patterns. We employ our method in the inference pipeline of ProbLog and conduct extensive experiments. We show that our compaction method improves knowledge compilation and consecutively the overall inference performance. Furthermore, using compaction reduces the number of time-outs, allowing us to solve previously unsolvable problems.

1 Introduction

Knowledge compilation [6] encompasses a set of methods to compile a Boolean formula for which some inference tasks are computationally expensive into a Negation Normal Form (NNF) with special properties that allow to solve the same tasks efficiently. Knowledge compilation finds application in planning [21], computer-aided design [20], probabilistic reasoning [6,8]. Even if solving those problems on the compiled Boolean formulae is efficient, knowledge compilation itself is an #P-complete problem [27].

State-of-the-art Probabilistic Logic Programming systems like ProbLog [7,11] use knowledge compilation approaches to reduce the expensive inference task to a weighted model counting (WMC) problem. Motivated to solve larger problems in ProbLog, in this paper we present an optimization method that compacts Boolean formulae in order to speed-up knowledge compilation. While we implemented our approach in the scope of ProbLog and used common ProbLog problems to evaluate its effectiveness our approach is more general and any application using Boolean formulae to represent knowledge could benefit from it.

F. Calimeri et al. (Eds.): LPNMR 2015, LNAI 9345, pp. 425–438, 2015.
DOI: 10.1007/978-3-319-23264-5_35

Our first contribution is the identification of seven Boolean subformulae patterns that can be detected and used to re-write Boolean formulae in order to improve knowledge compilation. Our detected patterns fall into two types: one type that retains equivalence with the input Boolean formulae and a second type that reduces the number of Boolean variables contained in the formulae. The latter type patterns correspond to AND/OR clusters [16]. While they do not preserve the equivalence directly, an application specific equivalence can be defined and computed. In the context of ProbLog we preserve the weighted model count of the Boolean formulae.

Our second contribution is the implementation of an efficient algorithm that detects and compacts the presented patterns. Our implementation is independent from any ProbLog system. We incorporated it in two different implementations of ProbLog: MetaProbLog [17] and ProbLog2 [8] and evaluated it extensively with 7 different benchmarks. Further than the empirical evaluation of our approach we also provide a complexity analysis that shows that our algorithm is polynomial.

This paper builds on and extends the work presented in [25]. We introduced two new patterns, namely the minimal proof and OR-Cluster II; we improved the performance of the implementation; allowed it to work with multiple queries and evidence; and performed extensive experiments within the scope of ProbLog.

The paper is structured as follows: in Sect. 2 we present background and discuss related work; Sect. 3 describes the patterns while Sect. 4 gives an overview of the algorithm we implemented to detect and compact them; in Sect. 5 we analyze the effects of our compaction on inference with ProbLog; finally, we present our conclusions in Sect. 7.

2 Background

2.1 The Probabilistic Logic Programming Language ProbLog

ProbLog [7,11] is a general purpose Probabilistic Logic Programming (PLP) 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. No probabilistic fact can unify with a head of a rule in a ProbLog program. A simple ProbLog program is shown in Example 1.

Example 1. The following ProbLog program encodes a probabilistic graph. The predicate e/2 encodes a probabilistic edge between two nodes; the predicate p/2 defines a path between nodes.

```
0.6::e(a, b).  0.3::e(a, d).  0.8::e(b, c).  0.2::e(e, f).     p(X, Y):- e(X, Y).
0.7::e(c, d).  0.4::e(d, f).  0.4::e(d, e).                    p(X, Y):- e(X, X1), p(X1, Y).
```

For a ProbLog program L each ground probabilistic fact[1] f_i can be *true* with probability p_i or *false* with probability $(1 - p_i)$. A particular decision d on

[1] Probabilistic facts can be ground or non-ground. [11] proves that finitely many groundings of non-ground probabilistic facts are sufficient to compute probabilities. That is why we restrict our discussion to programs with ground probabilistic facts.

the truth values of all probabilistic facts determines a unique logic program L^d. For N probabilistic facts there exist 2^N such logic programs. Each probabilistic fact can be seen as an independent random variable. ProbLog then defines a distribution over the logic programs L^d as:

$$P(L^d) = \prod_{f_i \in L^d} p_i \prod_{f_i \in L \setminus L^d} (1 - p_i) \quad (1)$$

ProbLog systems[2] provide a wide choice of inference and learning algorithms, which are used in applications like system prognostics and diagnostics [28], link and node prediction in biological data [10], robotics [19]. ProbLog focuses on two main inference tasks: (a) computing the probability that a query is true for a given ProbLog program, namely the marginal (MARG) probability of a query; and (b) computing the conditional probability (COND) of a query for a ProbLog program given some evidence, i.e. a set of facts for which the truth values have been decided.

Computing the MARG task boils down to determining all logic programs L^d which entail the query and summing their probabilities as computed by Eq. 1. Similar, for the COND task ProbLog needs to determine the logic programs L^d which entail the query but also the evidence. An exhaustive enumeration of these programs is infeasible but for the tiniest problems. That is why ProbLog's inference mechanism employs a step-wise procedure called an *inference pipeline* [26] that reduces the inference task into a WMC problem. First, given a ProbLog program L, a set of queries and evidence, ProbLog uses SLD [12] or SLG [4] resolution on the logical part of L, that is, ignoring the label of probabilistic facts, in order to determine the ground logic program relevant to the queries and the evidence (Ground LP) [8]. Then, the Ground LP is converted to an equivalent with respect to the WMC Boolean formula. During this process any cycles that occur in the Ground LP are handled. We use the *Proof-Based* approach [15] for this. Next, using knowledge compilation the Boolean formula is compiled into a negation normal form (NNF) with special properties which allows efficient WMC. Two target compilation languages have been considered so far: *ROBDDs* [3] and *sd-DNNFs* [6]. The NNF is then associated with the probabilities of L and used for WMC.

2.2 AND-OR Graphs

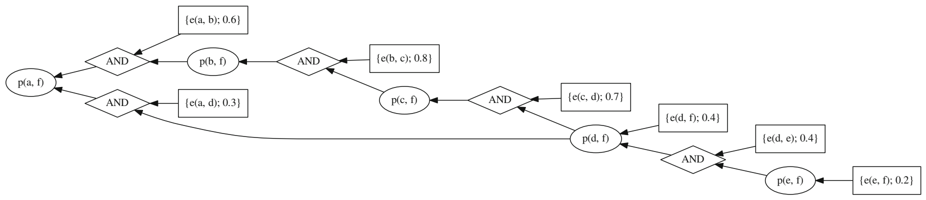
We represent Boolean formulae as AND-OR graphs. An AND-OR graph is a directed graph composed by AND and OR nodes. An AND node indicates that all child nodes must be true, while an OR node indicates that at least one of the child nodes must be true. An AND-OR graph is a suitable representation for a ground logic program relative to a query q. The different clauses ($q_{i \in 1..m}$:- $r_{i,1}, ..., r_{i,n}$.) of the predicate q are processed as follows: for each clause q_i all literals $r_{i,j}$ in the body are grouped as children of an AND node. The different AND nodes then are grouped as children of an OR node labeled with q. Next,

[2] When it is clear from the context we use the term ProbLog to refer to either the language or the system. Otherwise we state it explicitly.

each literal $r_{i,j}$ is treated as a new query. An AND-OR graph of a query has the following characteristics: cycles that appear in the logic program also appear in the AND-OR graph; for each subgoal g there is only one OR node; an OR-node has multiple parents if the subgoal is repeated and goals proven as facts are represented by special OR nodes without children, called terminal nodes. The edge from a child node to a parent node states that the parent depends on the child node.

Definition 1. *An **AND-OR graph** for a query q is a directed graph $G = (V_{and}, V_{or}, V_{term}, E)$ with V_{and} a set of AND nodes, V_{or} a set of labeled OR nodes, $V_{term} \subset V_{or}$ a set of terminal nodes, $V_{nonterm} = V_{or} \setminus V_{term}$ and $E \subseteq R$ a set of directed edges, where $R = (V_{and} \times V_{or}) \cup (V_{nonterm} \times V_{and}) \cup (V_{nonterm} \times V_{or})$. The root of the graph is an OR node labeled with q.*

Example 2. For the ProbLog program in Example 1 and the query `p(a, f).` the corresponding AND-OR graph is:



Ellipses depict OR nodes, diamonds depict AND nodes and rectangles terminal nodes. OR nodes are labeled with the goal they prove. Note that in the context of ProbLog terminal nodes have attached probabilities.

3 Compactable Patterns

We identify 7 patterns that appear in AND-OR graphs and present how we use them in order to compact the graph. The patterns we identify and their compacted form are illustrated in Table 1. Patterns 1 to 4 maintain the Boolean formulae equivalence. The compaction of patterns 5 to 7 removes Boolean variables and introduces a new Boolean variable to represent them. These compactions do not directly maintain the equivalence of the Boolean formulae. Application specific problems require a special calculation for the introduced representative Boolean variable. For correct ProbLog inference we need to maintain the WMC. That requires to calculate the probability of the representative Boolean variable. Proof of correctness for these compactions appears in [16].

1. **Single Variable**: There are an OR node A and a terminal node B, such that A depends only on B. **Compaction**: Node A and the edge from B to A are deleted. The edges starting from A now start from B.
2. **Single Branch I**: There are a node A, an OR node B and an AND node C, such that B depends only on C and A depends on B. **Compaction**: If A is an OR node then node B and the edge from C to B are deleted. A new edge

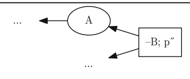
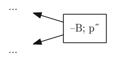
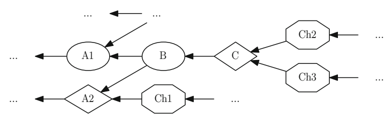
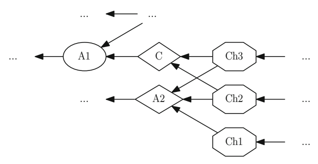
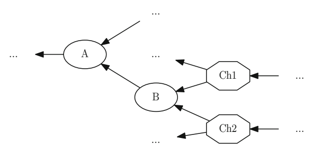
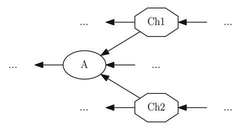
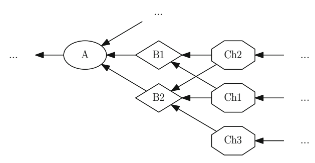
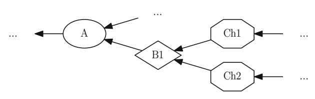
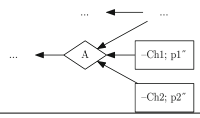
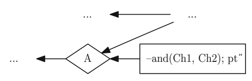
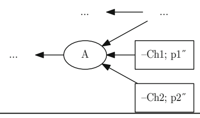
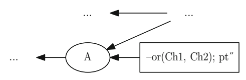
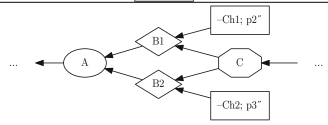
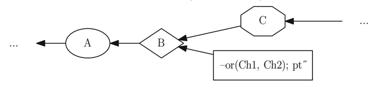
from C to A is created. If A is an AND node then nodes B and C are deleted together with the edge from C to B. All children of C are connected to A.

3. **Single Branch II**: There are two OR nodes A and B, such that A depends on B and no other node depends on B. **Compaction**: Node B and the edge from B to A are deleted. All children of B are connected to A.
4. **Minimal Proof**: There are an OR node A, two AND nodes B_1 with a set of children Ch_{B_1} and B_2 with a set of children Ch_{B_2} such that $Ch_{B_1} \subseteq Ch_{B_2}$. **Compaction**: Node B_2 and all edges from the child nodes in Ch_{B_2} to B_2 are deleted. The edge from B_2 to A is deleted as well.
5. **AND-Cluster**: There are an AND node A, a set of nodes $Ch'_A \subseteq Ch_A$, where Ch_A are all terminal nodes A depends on, such that $Ch'_A = Ch_A \setminus \{C|\exists B, B \neq A, B \text{ depends on } C\}$. **Compaction**: All terminal nodes $C_i \in Ch'_A$ are deleted, together with the edges from C_i to A. A new terminal node C_t is created together with an edge from C_t to A. A joint probability $p_t = \prod_{C_i \in Ch'_A} p_i$, where C_i is a terminal node with probabilistic label p_i is calculated. The probabilistic label p_t is attached to node C_t.
6. **OR-Cluster I**: There are an OR node A, a set of nodes $Ch'_A \subseteq Ch_A$, where Ch_A are all terminal nodes A depends on, such that $Ch'_A = Ch_A \setminus \{C|\exists B, B \neq A, B \text{ depends on } C\}$. **Compaction**: All terminal nodes $C_i \in Ch'_A$ are deleted, together with the edges from C_i to A. A new terminal node C_t is created together with an edge from C_t to A. A joint probability p_t is calculated as $p_t = (..((p_1 * (1 - p_2) + p_2) * (1 - p_3) + p_3)..+ p_n)$, where p_i is the probability labeled in $C_i \in Ch'_A$, $i = 1..|Ch'_A|$. The probability p_t is attached to node C_t.
7. **OR-Cluster II**: There are an OR node A, that depends on n AND nodes $B_1...B_n$ that each has exactly one different terminal child node $Ch_1...Ch_n$ and all the rest child nodes (denoted as node C) are common. **Compaction**: All AND nodes $B_1...B_n$ and all terminal nodes $Ch_1...Ch_n$ are deleted. A new terminal node Ch is created. A joint probability p_t is calculated as $p_t = (..((p_1 * (1 - p_2) + p_2) * (1 - p_3) + p_3)..+ p_n)$, where p_i is probabilistic part of the label of $Ch_i, i = 1..n$. The probabilistic label p_t is attached to node Ch. A new AND node B that contains Ch, C is created, finally, an edge from B to A is created.

4 Algorithm

Our algorithm iterates over patterns 1 to 6 in the order presented in Table 1. As soon as a pattern is detected the corresponding compaction is applied. According to the order we choose the detection and compaction of one pattern allows the detection and compaction of the next one in the same iteration. This ensures the minimum number of iterations required to compact a graph. Our algorithm terminates once no patterns can be detected. Algorithm 1 presents the pseudo-code for detecting patterns 1 to 6.

Table 1. AND-OR graph patterns and the compacting transformations. We denote with "..." multiple possible nodes to/from which exists an edge. With octagons we represent nodes that can be of any type (terminal, AND or OR).

	Pattern	**Compaction**
1.		
2.		
3.		
4.		
5.		$p_t = p_1 \cdot p_2$ 
6.		$p_t = p_1 \cdot (1 - p_2) + p_2$ 
7.		$p_t = p_1 \cdot (1 - p_2) + p_2$ 

Completeness: Our algorithm does neither detect nor compact pattern 7. We are also confident that there exist more patterns which we do not consider. Thus, AND-OR graphs which include at least one of these patterns will not be fully compacted. Therefore, our algorithm is not complete.

```
Data: An AND-OR graph
Result: Detected Node, Nodes to be compacted
detect_single_variable(NodeA, Terminal) ←
   or_edge(NodeA, Terminal),
   is_terminal(Terminal),
   ∄ and_edge(NodeA, _),
   ∄ (or_edge(NodeA, Any), Terminal ≠ Any).
detect_single_branch1(NodeB, NodeC) ←
   or_edge(NodeB, NodeC),
   and_edge(NodeC, _),
   ∄ and_edge(NodeB, _),
   ∄ (or_edge(NodeB, Any), NodeC ≠ Any).
detect_single_branch2(NodeA, NodeB) ←
   or_edge(NodeA, NodeB),
   or_edge(NodeB, _),
   ∄ and_edge(_, NodeB),
   ∄ (or_edge(Any, NodeB), NodeA ≠ Any).
detect_minimal_proof(NodeB1, NodeB2) ←
   or_edge(NodeA, NodeB1),
   and_edge(NodeB1, _),
   or_edge(NodeA, NodeB2),
   and_edge(NodeB2, _),
   NodeB1 ≠ NodeB2,
   all(Child, and_edge(NodeB1, Child),
                                        ChildsB1),
   all(Child, and_edge(NodeB2, Child),
                                        ChildsB2),
   $ChildsB1 \subseteq ChildsB2$.
detect_and_cluster(RefChilds) ←
   and_edge(NodeA, _),
   all(Terminal, (
     and_edge(NodeA, Terminal),
     is_terminal(Terminal),
     ∄ or_edge(_, Terminal)
   ), Childs),
   get_all_and_edge_sets(ChildSets),
   refine_cluster(ChildSets, Childs, RefChilds),
   $RefChilds \neq \emptyset$.
detect_or_cluster1(RefChilds) ←
   or_edge(NodeA, _),
   all(Terminal, (
     or_edge(NodeA, Terminal),
     is_terminal(Terminal),
     ∄ and_edge(_, Terminal)
   ), Childs),
   get_all_or_edge_sets(ChildSets),
   refine_cluster(ChildSets, Childs, RefChilds),
   $RefChilds \neq \emptyset$.
refine_cluster([], RefChilds, RefChilds)
refine_cluster([Set|ChildSets], Childs,
                                   RefChilds) ←
   $NewChilds = Set \wedge Childs$ ,
   refine_cluster(ChildSets, NewChilds,
                                   RefChilds).
```

Algorithm 1: The 6 pattern detection algorithms.

Complexity:[3] The compaction operations are very efficient ($O(N)$ with N the number of edges affected). Detecting and verifying a pattern is computationally expensive and deserves a thorough analysis.

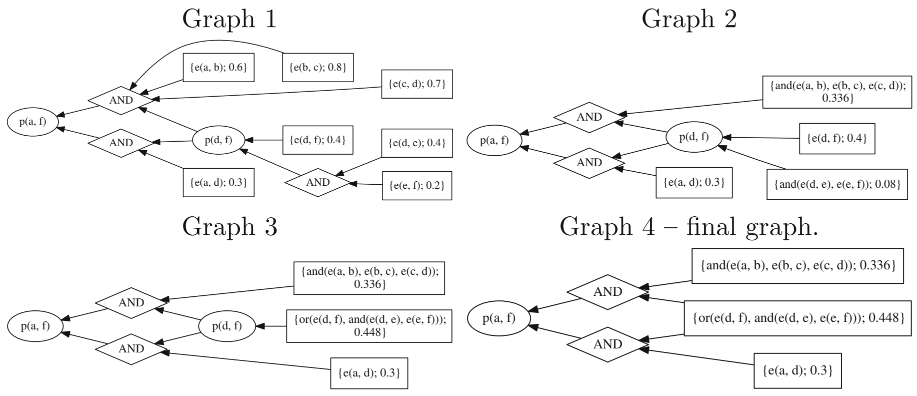
For an arbitrary AND-OR graph G we denote with N_{or} the number of *OR edges*, with N_{and} the number of *AND edges* and with N_{term} the number of terminal nodes. We assume that a node always contains N_{term} children; this is a high upper bound assumption but does not affect the complexity class.

The complexity for detecting and verifying all patterns 1 to 3 in an AND-OR graph is $O(N_{or} \cdot (log(N_{or}) + log(N_{and})))$; for all patterns 4 the complexity is $O(N_{or} \cdot (log(N_{or}) + log(N_{and}) + N_{term}))$; for all patterns 5 the complexity is $O(N_{and}^2 \cdot N_{term})$; finally, for all patterns 6 the complexity is $O(N_{or}^2 \cdot N_{term})$.

We illustrate the steps taken when applying our compaction algorithm to an AND-OR graph derived from the ProbLog program in Example 3.

Example 3. We apply our compaction algorithm on the graph in Example 2. In the 1st iteration it detects 1 **Single Variable** of $p(e, f)$ and 2 **Single Branch I** of $p(b, f)$ and $p(c, f)$ resulting in Graph 1 in the following table; and 2 AND-Clusters resulting in Graph 2. In the 2nd iteration 1 **OR-Cluster I** and 1 **Single Variable** of $p(d, f)$ are detected resulting in Graph 3 and Graph 4 accordingly.

[3] More details for the algorithm and the full complexity analysis can be found at: https://lirias.kuleuven.be/bitstream/123456789/500398/5/appendix.pdf.



The final AND-OR graph forms 1 **OR-Cluster II** pattern. If we detected and compacted OR-Cluster II patterns, it would enable a final AND-Cluster compaction to fully compact the AND-OR graph into a single terminal node containing the probability of the query.

The implementation neither detects nor compacts pattern 7; pattern 7 may correspond to complex subgraphs with unreasonably high detection cost. By using indexing we decreased the complexity of our previous implementation [25] from $O(N^2)$ to $O(N \cdot log(N))$ for several tasks. We also added support for multiple queries and evidence. The implementation of the detection/compaction algorithm is a stand-alone Prolog program.

5 Compacting ProbLog Programs

Section 2.1 presents the general scheme of a ProbLog inference pipeline. We focus on 4 particular ProbLog pipelines, based on two mainstream ProbLog implementations – MetaProbLog [17] and ProbLog2 [8]. These inference pipelines differ with respect to (a) representation of the Ground LP and the Boolean formulae: ProbLog2 uses AND-OR graphs and CNF DIMACS, while MetaProbLog uses Nested Tries [15] and BDD scripts [14]; (b) ways of preprocessing the Boolean formulae: ProbLog2 uses Boolean subformulae repetition detection and MetaProbLog uses the recursive node merging method presented in [18]; and (c) in the knowledge compilation method: ProbLog2 uses the sd-DNNF compiler *c2d* [5] and MetaProbLog uses the SimpleCUDD [14] compiler for ROBDDs. The 4 pipelines we use for our experiments are listed in Table 2. The pipeline implementations of ProbLog allow us to employ our detection/compaction algorithm (a) before and (b) after the cycle handling processing of the Boolean formula in any ProbLog pipeline. In (a), called the *prior* compaction, the Ground LP is represented as an AND-OR graph and then processed by our algorithm. In ProbLog2 the loop-breaking mechanism applies directly on the AND-OR graph and generates a loop-free AND-OR graph. In MetaProbLog the loop-breaking operates on the nested trie structure and produces a BDD Script which is easily rewritten as an AND-OR graph. This allows (b), that is, to invoke the compaction algorithm again and attempt a further optimization of the AND-OR graph before the knowledge compilation step. We call this the *post* compaction. Furthermore, we can invoke the *prior* and *post* compactions in the same pipeline; we refer to this compaction setting as *both*.

Table 2. ProbLog pipelines.

Pipeline	Ground LP representation	Cycle handling	Boolean formulae representation	Compilation language
ProbLog2/sd-DNNF[a]	AND-OR	Proof-Based	AND-OR→CNF DIMACS	sd-DNNF
ProbLog2/ROBDD	AND-OR	Proof-Based	AND-OR→BDD script	ROBDD
MetaProbLog/sd-DNNF	Nested Tries	Proof-Based + [18]	BDD script→CNF DIMACS	sd-DNNF
MetaProbLog/ROBDD[b]	Nested Tries	Proof-Based + [18]	BDD script	ROBDD

[a]ProbLog2 and [b]MetaProbLog default pipelines.

5.1 Experimental Set-Up

We experiment with 7 benchmark sets with a total of 738 programs. These benchmark sets have been previously used for testing the performance of different ProbLog implementations. The variety of these benchmarks and the different inference tasks ensure a realistic estimate of the gain or the loss in the performance of ProbLog pipelines due to our compaction algorithm.

In order to present our data in a more comprehensive way, we divide our benchmarks in three groups: 387 *easy* programs which consume less than 10 s; 99 *medium* programs which consume between 10 and 60 s and 150 *hard* programs which consume more than 60 s. To classify a program we use the total run time for the MetaProbLog/ROBDD pipeline without compaction – the default MetaProbLog pipeline.

Each program is executed with the 4 ProbLog pipelines and the 4 compaction settings – *none*, *prior*, *post* and *both*. Their combination results in 16 different ProbLog pipelines to run each benchmark program with. We chose a time-out of 540 s for each test run. We managed to solve 636 out of the 738 programs within the 540 s time-out with at least one of the 16 pipelines.

The c2d compiler is non-deterministic [5], meaning that for the same CNF the compiled sd-DNNFs may differ. That is why we run each test invoking c2d 5 times (8 pipelines use c2d). Then we report the average time consumed by the test. From previous experiments we have determined this number to give a reliable estimate of the performance of c2d.

We run our experiments on 17 computers with Intel® quad-core 64-bit CPU at 2.83GHz, 8GBs of RAM running Ubuntu 12.04 LTS (under normal load). The chosen time-out ensured our experiments to terminate within at most 14 days.

5.2 Experimental Results

In our experiments we collect the *total* run time for executing a benchmark program (including the compaction time). We use the time results to determine

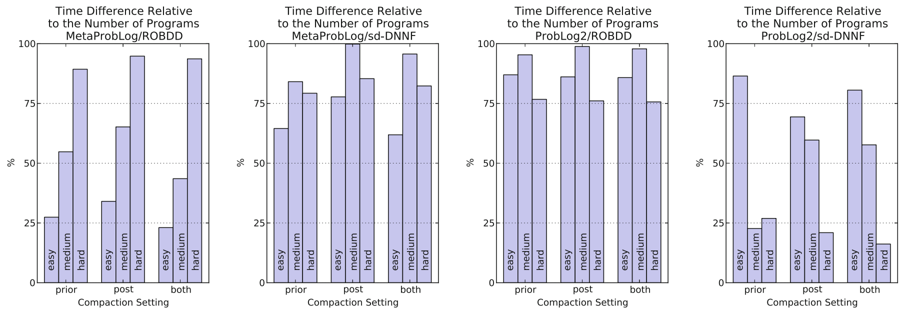


Fig. 1. Relative time gain due to a specific compaction.

the compaction setting which leads to (a) the lowest run times; and (b) to the lowest number of time-outs for each of the pipelines and each benchmark set.

For each ProbLog pipeline and each compaction $c \in \{prior, post, both\}$ we sum (a) the gain in the total run time for each benchmark compared with the run time of no compaction ($c = none$) when the compaction performs better: T^g_C (this is the total gain for compaction); (b) the gain in the total run time when no compaction performs better: T^g_N (this is the total loss for compaction). We normalize each gain by dividing by the total number of programs within a benchmark set to compensate for the fact that some of them contain more programs. For example, consider a particular benchmark with programs $\{b_1, b_2, b_3\}$ each with run times: with compaction 20, 30 and 40 s and with no compaction 10, 25, 70 s. Then $T^g_C = 70 - 40 = 30$ and $T^g_N = (20 - 10) + (30 - 25) = 15$ showing that in total the gain with compaction exceeds the loss due to compaction. We exclude programs for which both inference with compaction and with no compaction times out. We chose this measurement because it shows the overall gain in run time due to compaction. We present the gain due to compaction relative to the total gain $\frac{T^g_C}{T^g_C + T^g_N}$ in Fig. 1 as percentage. Detailed results are given in our online appendix. We base our conclusions on all the results.

5.3 Experimental Conclusions

First, our algorithm improves the knowledge compilation time for the majority of the benchmarks, between 75 % to 85 % for ROBDDs and between 55 % to 65 % for sd-DNNFs. Our intuition is that ROBDDs benefit more than sd-DNNFs because ROBDDs use a general Boolean formula for input while sd-DNNFs require a conversion to a CNF Boolean formula. Compaction is beneficial for most of the medium and hard problems but not that much for the easy problems. It was expected that the time spend to perform our algorithm would not be compensated from the gain in knowledge compilation for small problems.

Second, regarding the used pipelines from Fig. 1 we conclude that the highest gain from our algorithm was for the ProbLog2/ROBDD pipeline having an almost 100 % gain for all compaction settings (on medium problems). Using compaction is preferable to no compaction for the MetaProbLog/ROBDD, MetaProbLog/sd-DNNF and ProbLog2/ROBDD pipelines and complex problems. We also note that the ProbLog2/sd-DNNF pipeline in overall does not benefit from our approach.

Third, none of the compaction settings (i.e. the *prior*, *post* or *both*) outperforms the other compaction settings. For MetaProbLog/ROBDD the *both* compaction is preferable; for MetaProbLog/sd-DNNF and ProbLog2/ROBDD pipelines preferable is the *post* compaction; for ProbLog2/sd-DNNF the *prior* compaction. The *post* and *both* compactions often yield the same Boolean formulae. In such cases *both* spends unnecessary extra time for *prior* compaction. We also note that the actual cost to perform detection and compaction is generally small. In particular, compacting AND-OR graphs generated from a Ground LP consumed at most 18.25 s for a program with total run time of 264.27 s; compacting AND-OR graphs generated from Nested Tries consumed at most 64.95 s for a program with total run time of 297.79 s.

Fourth, compaction allowed us to solve significantly more problems that would otherwise timeout. Particularly, in the best case, MetaProbLog/ROBDD with *both* compaction, we can solve 38 % more programs; ProbLog2/ROBDD with *post* compaction can solve 37 % more programs. The two pipelines which use compilation to sd-DNNF benefit less from compaction, MetaProbLog/sd-DNNF with *post* compaction can solve 6 % more programs while for ProbLog2/sd-DNNF compaction introduces up to 12 % more timeouts. The extra time-outs occur for benchmarks that contain multiple queries and evidence. Often the query and evidence atoms appear also as subgoals. Queries and evidence are required for the final step of WMC thus they should not be removed from the Boolean formula. Therefore there are less patterns that can be compacted in the case of COND with respect to MARG tasks. For the other benchmarks compaction reduces the overall amount of time-outs.

Finally, following from all our results, we must indicate that there is not one best performing pipeline over all benchmarks. On average, the pipeline with the least timeouts was ProbLog2/ROBDD with *post* compaction. The gain due to compaction (*prior*, *post* or *both*) on the hard problems and the decrease of timeouts indicate that our approach improves the performance of the system at problems that it was poor before.

6 Related Work

Rewriting a Boolean formula to improve the performance of knowledge compilation in the scope of ProbLog had first been investigated in [18]. [18] shows that feeding a rewritten Boolean formulae instead of a non optimized one reduces the operations needed by the knowledge compilation step and consequently the knowledge compilation run time. The work we present in this paper, focuses on optimizing even further the Boolean formula and works in parallel with these Boolean formulae rewrites. Boolean formulae rewriting, in the scope of assessing the Probability of a Sum-of-Products, has been investigated also in [24].

Detecting regularities such as AND/OR-Clusters on a Boolean formula in normal form (e.g., DNF), has been investigated in [16]. Our approach performs similar transformations on an AND-OR graph instead of a Boolean formula in normal form. [16] proves completeness of detecting AND/OR-Clusters in Boolean formulae but faces some practical limitations. The most important of which is that ProbLog using *tabling* and *cycle handling* as presented in [15] would generate a Boolean formula that is not in normal form.

Hintsanen [9] argues that structural properties are important for finding the most reliable subgraph. He calculates the probability of subgraphs connecting two nodes and searching for the subgraph with the maximum probability. The paper identifies as a special case the series-parallel subgraphs for which they can compute the probability polynomially. These series-parallel subgraphs have similarities with the AND/OR-Clusters.

Our work is also similar to [13] which presents a preprocessing of propositional formulae to optimize model counting. Their approach optimizes CNF Boolean formulae by using seven preprocessing methods. Similar to our work, some of their preprocessing methods maintain equivalence and others not. In contrast to our approach some of their methods increase the size of the Boolean formulae which is an interesting point for us to look upon. There exist several other related works from other fields such as variable ordering approaches for BDDs [22,23] or preprocessing methods used in SAT solving [1,2].

7 Conclusion and Future Work

This paper presented a pattern-based approach for compacting Boolean formulae. It detects and compacts 6 (out of 7 identified) patterns – 4 that preserve logic equivalence and 2 that preserve equivalence with respect to the weighted model count. Our approach aims to improve probabilistic inference that uses knowledge compilation and weighted model counting. It targets but is not limited to the probabilistic logic programming system ProbLog and its underlying implementations.

We performed experiments with 4 different ProbLog pipelines and 3 compaction settings on 7 benchmark sets with 738 benchmarks in total. Our results show that compaction improves knowledge compilation to ROBDDs as well as to sd-DNNFs. The gain in the total run time due to compaction is most salient for harder problems. The decreased amount of time-outs proves that our approach enables inference on problems unsolved before (i.e. without compaction).

In the future we want to investigate also non-compacting transformations that could aid (thus improve) the knowledge compilation. In addition, we plan to extend our algorithm to handle problems outside the domain of ProbLog. We aim to test it on benchmarks from [13] in order to determine its general effects.

Acknowledgments. We want to thank the anonymous reviewers for their comments and help to improve our paper. Theofrastos Mantadelis is funded by the Portuguese

Foundation for Science and Technology (FCT) within the projects SIBILA NORTE-07-0124-FEDER-000059 and UID/EEA/50014/2013. Dimitar Shterionov is funded by KU Leuven within the project GOA 13/010.

References

1. Aloul, F.A., Markov, I.L., Sakallah, K.A.: Faster SAT and smaller BDDs via common function structure. In: ICCAD, pp. 443–448 (2001)
2. Bacchus, F., Winter, J.: Effective preprocessing with hyper-resolution and equality reduction. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 341–355. Springer, Heidelberg (2004)
3. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986)
4. Chen, W., Swift, T., Warren, D.S.: Efficient top-down computation of queries under the well-founded semantics. JLP **24**(3), 161–199 (1995)
5. Darwiche, A.: A compiler for deterministic, decomposable negation normal form. In: AAAI/IAAI, pp. 627–634. AAAI Press/MIT Press (2002)
6. Darwiche, A., Marquis, P.: A knowledge compilation map. JAIR **17**, 229–264 (2002)
7. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: a probabilistic Prolog and its application in link discovery. In: IJCAI, pp. 2468–2473. AAAI Press (2007)
8. Fierens, D., Van den Broeck, G., Renkens, J., Shterionov, D.S., Gutmann, B., Thon, I., Janssens, G., De Raedt, L.: Inference and learning in probabilistic logic programs using weighted boolean formulas. TPLP **15**(3), 358–401 (2015)
9. Hintsanen, P.: The most reliable subgraph problem. In: Kok, J.N., Koronacki, J., Lopez de Mantaras, R., Matwin, S., Mladenič, D., Skowron, A. (eds.) PKDD 2007. LNCS (LNAI), vol. 4702, pp. 471–478. Springer, Heidelberg (2007)
10. Kimmig, A., Costa, F.: Link and node prediction in metabolic networks with probabilistic logic. In: Berthold, M.R. (ed.) Bisociative Knowledge Discovery. LNCS, vol. 7250, pp. 407–426. Springer, Heidelberg (2012)
11. Kimmig, A., Demoen, B., De Raedt, L., Santos Costa, V., Rocha, R.: On the implementation of the probabilistic logic programming language ProbLog. TPLP **11**, 235–262 (2011)
12. Kowalski, R.A.: Predicate logic as programming language. In: IFIP Congress, pp. 569–574 (1974)
13. Lagniez, J., Marquis, P.: Preprocessing for propositional model counting. In: AAAI, pp. 2688–2694 (2014)
14. Mantadelis, T., Demoen, B., Janssens, G.: A simplified fast interface for the use of CUDD for binary decision diagrams (2008). http://people.cs.kuleuven.be/theofrastos.mantadelis/tools/simplecudd.html
15. Mantadelis, T., Janssens, G.: Dedicated tabling for a probabilistic setting. In: ICLP (Technical Communications), LIPIcs, vol. 7, pp. 124–133 (2010)
16. Mantadelis, T., Janssens, G.: Variable compression in ProbLog. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 504–518. Springer, Heidelberg (2010)
17. Mantadelis, T., Janssens, G.: Nesting probabilistic inference (2011). CoRR, abs/1112.3785
18. Mantadelis, T., Rocha, R., Kimmig, A., Janssens, G.: Preprocessing boolean formulae for BDDs in a probabilistic context. In: Janhunen, T., Niemelä, I. (eds.) JELIA 2010. LNCS, vol. 6341, pp. 260–272. Springer, Heidelberg (2010)

19. Moldovan, B., van Otterlo, M., Moreno, P., Santos-Victor, J., De Raedt, L.: Statistical relational learning of object affordances for robotic manipulation. In: ILP, p. 6 (2012)
20. Mostow, J.: Towards automated development of specialized algorithms for design synthesis: knowledge compilation as an approach to computer-aided design. Res. Eng. Des. **1**(3–4), 167–186 (1990)
21. Namioka, Y., Tanaka, T.: Knowledge compilation for interactive design of sequence control programs. In: IEA/AIE, pp. 363–368 (1996)
22. Narodytska, N., Walsh, T.: Constraint and variable ordering heuristics for compiling configuration problems. In: IJCAI, pp. 149–154 (2007)
23. Panda, S., Somenzi, F.: Who are the variables in your neighborhood. In: ICCAD, pp. 74–77 (1995)
24. Rauzy, A., Châtelet, E., Dutuit, Y., Bérenguer, C.: A practical comparison of methods to assess sum-of-products. Reliab. Eng. Syst. Saf. **79**(1), 33–42 (2003)
25. Shterionov, D., Mantadelis, T., Janssens, G.: Pattern-based compaction for ProbLog inference. In: ICLP (Technical Communications), pp. 1–4 (2013)
26. Shterionov, D., Janssens, G.: Implementation and performance of probabilistic inference pipelines. In: Pontelli, E., Son, T.C. (eds.) PADL 2015. LNCS, vol. 9131, pp. 90–104. Springer, Heidelberg (2015)
27. Valiant, L.G.: Why is Boolean complexity theory difficult? In: London Mathematical Society Symposium on Boolean Function Complexity, pp. 84–94 (1992)
28. Vlasselaer, J., Meert, W.: Statistical relational learning for prognostics. In: Belgian-Dutch Conference on Machine Learning, pp. 45–50 (2012)