Pattern-Based Compaction for ProbLog Inference

Dimitar Shterionov
Theofrastos Mantadelis
Gerda Janssens

Report CW 639, July 2013

KU Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Pattern-Based Compaction for ProbLog Inference

Dimitar Shterionov
Theofrastos Mantadelis
Gerda Janssens

Report CW 639, July 2013

Department of Computer Science, KU Leuven

Abstract

The probabilistic inference of current Probabilistic Logic Programming systems uses weighted model counting (WMC). To compute the success probability of a query, first the set of proofs of the query is computed. Next, the set of proofs is converted into a corresponding Boolean formula which is compiled into a suitable representation that allows WMC in polynomial time. The computational cost of each step depends on the size and the complexity of the output of the previous step. The set of proofs, computed in the first step, can become large. In this paper we propose a way to reduce the size of the collected proofs in order to improve the next steps. We use an AND-OR graph to represent the collected proofs. Our method compacts the AND-OR graph by detecting several patterns and applying transformations on them. Experiments confirm that by decreasing the size of the AND-OR graph, we reduce the time and memory consumption of the next steps. Our algorithm enables probabilistic inference on problems previously unsolvable.

Keywords : Boolean formulae compaction, variable compression, AND-OR graphs, AND-/OR- clusters, ProbLog2, MetaProbLog, pattern, subgraph, probabilistic inference.

Pattern-Based Compaction for ProbLog Inference

DIMITAR SHTERIONOV, THEOFRASTOS MANTADELIS, GERDA JANSSENS
Department of Computer Science, KU Leuven
(*e-mail:* `firstname.secondname@cs.kuleuven.be`)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

The probabilistic inference of current Probabilistic Logic Programming systems uses weighted model counting (WMC). To compute the success probability of a query, first the set of proofs of the query is computed. Next, the set of proofs is converted into a corresponding Boolean formula which is compiled into a suitable representation that allows WMC in polynomial time. The computational cost of each step depends on the size and the complexity of the output of the previous step. The set of proofs, computed in the first step, can become large. In this paper we propose a way to reduce the size of the collected proofs in order to improve the next steps. We use an AND-OR graph to represent the collected proofs. Our method compacts the AND-OR graph by detecting several patterns and applying transformations on them. Experiments confirm that by decreasing the size of the AND-OR graph, we reduce the time and memory consumption of the next steps. Our algorithm enables probabilistic inference on problems previously unsolvable.

1 Introduction

Combining logic and probabilities resulted in the field of Probabilistic Logic Programming (PLP). Probabilistic logic programs are logic programs in which some of the facts are annotated with probabilities. Each of these (probabilistic) facts can either be true or false. A specific decision on the truth values of all probabilistic facts is called a possible world. A probabilistic program gives rise to an exponential number of such possible worlds. The possible worlds take part in one of the key PLP inference tasks: determining the probability that a query succeeds. A query is true in a subset of the possible worlds of a PLP program. These are the models of the program with respect to the query. In the model each fact has a weight: its probability if the fact is true or (1− its probability) if the fact is false. The product of these weights is the weight of the model. The sum of all models' weights, known as *weighted model count* (WMC), is the success probability of a query. ProbLog (De Raedt et al. 2007) is a Prolog-based PLP system, which uses (a form of) WMC as its main inference mechanism.

Enumerating all possible worlds is a computationally expensive task. Current state-of-the-art probabilistic inference methods can avoid the exponential enumeration of the possible worlds (De Raedt et al. 2007; Fierens et al. 2011; Sato and Kameya 1997). These methods first collect successful proofs for the query. Then, the set of proofs are *converted* into a suitable Boolean formula representation on which WMC can be done in polynomial time. The latter step is well studied in the context of knowledge representation and logic programming (Janhunen 2004; Darwiche and Marquis 2002; Darwiche 2009) and resulted in advanced systems (Somenzi 2005; Darwiche 2004; Muise et al. 2012). Unfortunately the

limits of these systems are being reached during probabilistic inference: the conversion can take too long or the size of the Boolean formula can become too large. Crucial for the efficiency of the conversion are the size and the complexity of the proofs: the number of atoms and how they form conjunctions and disjunctions.

In this paper we aim at compacting the set of proofs in order to reduce the complexity of the conversion. We use this as an enabling operation. Proofs often share atoms which appear in the Boolean formula as repeating subformulae. When possible, a transformation of such a subformula to an equivalent but smaller-sized representation reduces the size of the Boolean formula. We know from our earlier work (Mantadelis and Janssens 2010) that compressing AND-/OR- clusters in DNFs significantly decreases the size of Boolean formulae and increases the overall performance of the PLP system. This method, though, has several drawbacks: (i) it requires the proofs to be converted to a Boolean formula in normal form (such as DNF) prior to detecting and optimizing the subformulae and (ii) this approach cannot handle negated Boolean (sub)formulae generated by PLP systems which support general negation, such as ProbLog.

The method which we introduce here is more general than the one presented in (Mantadelis and Janssens 2010). It operates on a more general representation of a Boolean formula – AND-OR graphs which facilitate the *pattern detection and compaction* and support negation. Therefore, our approach compacts the collected proofs before converting them to a Boolean formula in normal form (CNF or DNF) and handles negated atoms. Our algorithm preserves equivalence in terms of WMC. It is an enabling step that reduces the complexity of the generation of the Boolean formulae and thus decreases the execution time of the following steps of the inference. The AND-OR graph compaction can be incorporated in any PLP system. We implemented it and evaluated its impact for the two state-of-the-art ProbLog systems – MetaProbLog (Mantadelis 2012, Chapter 6) and ProbLog2 (Fierens et al. 2013). Our experiments show a compaction ratio of about 40%. Most of the time this compaction is beneficial for the conversion.

The paper is structured as follows. In Section 2 we introduce the AND-OR graph representation of the collected proofs. Section 3 describes the patterns and the corresponding compactions; in Section 4 we describe the algorithm and analyze its complexity. Experimental results are summarized in Section 5. The paper concludes with Section 6.

2 Background

2.1 MetaProbLog and ProbLog2

Both MetaProbLog and ProbLog2 are descendants of the original ProbLog system (De Raedt et al. 2007). MetaProbLog supports meta predicates and meta calls to probabilistic queries. Its inference algorithm represents the proofs by nested tries which are converted to a ROBDD (Bryant 1986). The success probability of a query is computed from the ROBDD.

ProbLog2 handles SRL tasks such as learning from interpretations. It represents the proofs by a ground ProbLog program which is transformed into a weighted CNF that is compiled into an sd-DNNF (Darwiche and Marquis 2002). From the sd-DNNF, ProbLog2 computes the conditional probabilities of multiple queries given some evidence.

Both MetaProbLog and ProbLog2 use SLG resolution to proof a query. MetaProbLog

stores the proofs as nested tries – a data structure that represents the SLG tree (Chen and Warren 1996) of a logic program, while ProbLog2 generates a ground probabilistic logic program (in short ground program).

2.2 AND-OR graph

In order to solve a problem, it can be decomposed into subproblems such that either all or only one need to be solved. The search space of the problem can be represented as a tree of AND and OR nodes – an AND-OR tree. AND nodes indicate that all children nodes have to be solved, while for an OR node only one of them needs to be solved. This is a very natural representation for the search space of a query in a logic program. A predicate p is defined by one or more clauses of the form $p :- q_1, .., q_n.$. To solve a query p, the different clauses of p are grouped in an OR node and all q_i literals in the body of a clause for p are grouped in an AND node. The set of proofs for a query is represented by an AND-OR graph: for each goal p there is only one OR node. A loop in a proof is represented by a loop in the graph. An OR-node can have multiple parents as it can appear in different proofs. The collected proofs of the ProbLog systems can easily be represented by an AND-OR graph. We use a specific form of an AND-OR graph with explicit terminal nodes to represent goals that are proven by facts and with edges between OR-nodes to avoid AND nodes with only one relevant child (see Figure 1). Our compaction is applied on such AND-OR graphs.

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 $R = (V_{and} * V_{or}) \cup (V_{nonterm} * V_{and}) \cup (V_{nonterm} * V_{or})$ and with as root the OR node with label q.

a. An edge connecting two OR nodes. b. An AND node with only one relevant child.

Fig. 1. OR-OR edges in an AND-OR graphs.

We say that a parent node *depends* on a child node. An edge points from a child to the parent. We depict an AND-OR graph with ellipses for OR nodes, diamonds for AND nodes and rectangles for terminal nodes. OR nodes are labeled with the goal they prove.

MetaProbLog stores the proofs in nested tries while ProbLog2 uses a ground program. Although these are two different representations they can equally easy be transformed into AND-OR graphs. In Figure 2 we give the AND-OR graph for the proofs of a query in a ProbLog program.

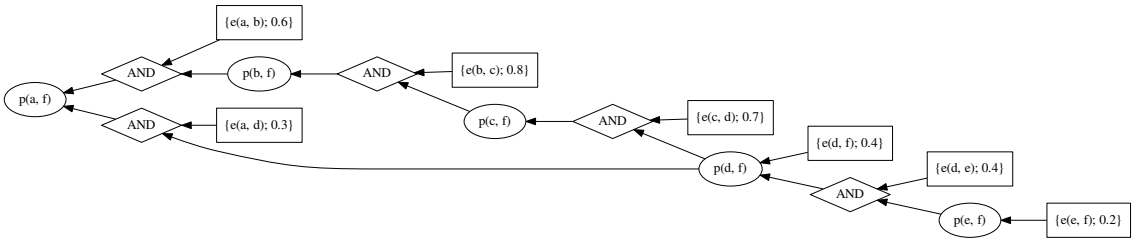
In the ProbLog program in Figure 2.a the `path/2` predicate (in short `p/2`) specifies that there exists a path in the probabilistic graph that is given by the `edge/2` (in short `e/2`) facts that associate with each edge a probability. ProbLog computes the probability that there exists a path from a to f. The set of four proofs is represented by the ground clauses for `p/2` in Figure 2.b. Note that the goal `p(d,f)` appears in the body of two

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

a. The initial ProbLog program and a query.

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

b. The ground ProbLog program.



c. The AND-OR graph for query `p(a,f)`.

Fig. 2. Set of proofs represented as a AND-OR graph.

different clauses. The AND-OR graph for the ground program is given in Figure 2.c. The root OR node for `p(a,f)` depends on two AND nodes which group the goals in the bodies of its two ground clauses. The probabilistic facts appear as terminal nodes.

When the edges define a cyclic graph, the ground program and the AND-OR graph can contain cycles. The next step of the probabilistic inference handles correctly the cycles by using well-known techniques as (Janhunen 2004) and (Mantadelis 2012, Chapter 3).

The aim of our method is to compact the AND-OR graph such that the next step in the inference becomes simpler. Consider two different AND-OR graphs $G_1 = (V^1_{and}, V^1_{or}, V^1_{term}, E_1)$ and $G_2 = (V^2_{and}, V^2_{or}, V^2_{term}, E_2)$ with query q as root, such that $|V^1_{and} \cup V^1_{or}| > |V^2_{and} \cup V^2_{or}|$ and G_2 is the result of a series of our pattern-based transformations applied on G_1 that preserve the WMC for query q. The next step converts G_1 and G_2 into the Boolean formulae, respectively BF_1 and BF_2, and $WMC(BF_1) = WMC(BF_2)$. The smaller-sized BF_2 could then be used to calculate the success probability of q instead of BF_1. In the next section we identify a set of useful patterns.

3 Patterns

Detecting regularities such as AND-/OR- clusters on a Boolean formula in normal form (e.g., DNF), has been investigated in (Mantadelis and Janssens 2010). Our goal is to apply similar transformations on an AND-OR graph before converting to a Boolean formula. We have observed that some regularities (such as the AND-/OR- clusters) form patterns in AND-OR graphs. Recognizing these *patterns* and *compacting* them efficiently are the two main tasks of our approach. Our method deals with the following patterns (see also Table 1):

1. Single Variable *Pattern*: 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 *Pattern*: 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 node 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 node 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 *Pattern*: 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. AND-Cluster I *Pattern*: an OR node A, an AND node B, a set of nodes $Ch'_B \subseteq Ch_B$, where Ch_B are all terminal nodes which B depends on, such that $Ch'_B = Ch_B \setminus \{D | \exists C, C \neq B, C \text{ depends on } D\}$. *Compaction*: all terminal nodes $C_i \in Ch'_B$ are deleted, together with the edges from C_i to B. A new terminal node C_t is created together with an edge from C_t to B. A joint probability $p_t = \prod_{C_i \in Ch'_B} p_i$, 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.
5. OR-Cluster I *Pattern*: an OR node A, a set of nodes $Ch'_A \subseteq Ch_A$, where Ch_A are all terminal nodes which 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 probabilistic part of the label of $C_i \in Ch'_A$, $i = 1..|Ch'_A|$. The probabilistic label p_t is attached to node C_t.
6. OR-Cluster II *Pattern*: An OR node A, two AND nodes B_1 and B_2 and a node (regardless the type) C, such that A depends on B_1 and on B_2. Also, B_1 and B_2 depend on C. This pattern is generalized for n AND nodes (B_1 to B_n) and for m common children nodes of B_1 to B_n (C_1 to C_m). *Compaction*: not dealt with.

Pattern 6 detects OR clusters which contain ANDs. It is computationally expensive to detect this pattern as it may have many variations, depending on the number of AND nodes, connected to the parent OR node and the number of child nodes, shared among the AND nodes. That is why currently we do neither detect nor compact this pattern.

Figure 3 illustrates the compacted AND-OR graph of our running example (Figure 2). The following patterns are detected: 2 times pattern 1, 2 times pattern 2, 3 AND clusters and 1 OR cluster. All the compaction steps are shown in Appendix A.

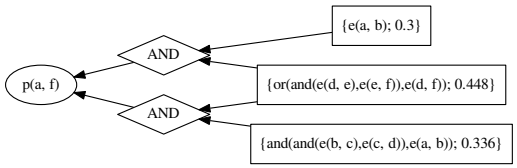


Fig. 3. The compacted AND-OR graph for the graph of Figure 2.c.

4 The Algorithm

Description. Our algorithm detects the patterns defined in Section 3 and compacts the AND-OR graph accordingly. In one iteration of the algorithm we go over all the patterns

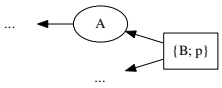
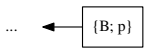
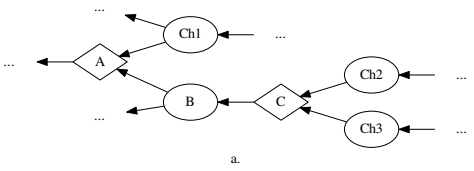
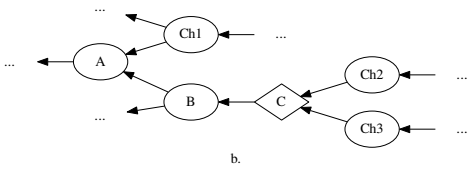
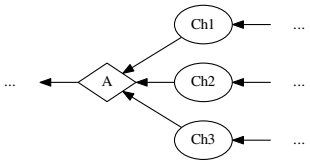
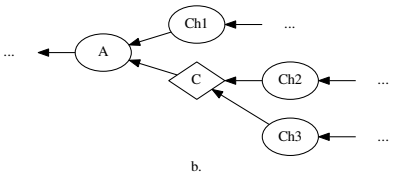
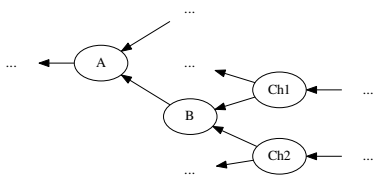
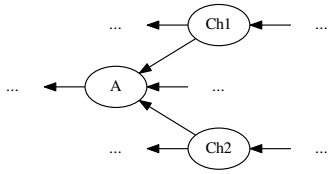
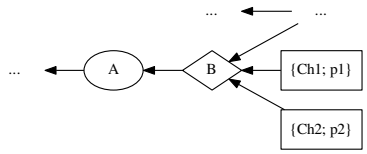
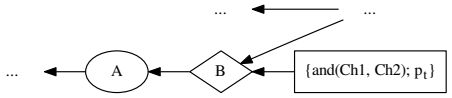
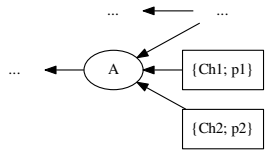
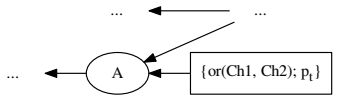
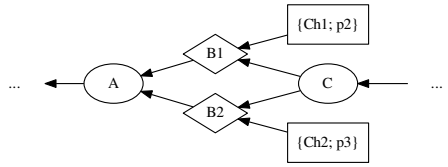
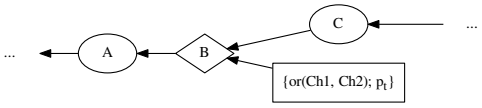
	Pattern	**Compaction**
1.		
2.	 a.  b.	 a.  b.
3.		
4.		 $p_t = p1 * p2$
5.		 $p_t = p1 * (1 - p2) + p2$
6.		 $p_t = p1 * (1 - p2) + p2$

Table 1. *AND-OR graph patterns and the compacting transformations. We denote with "..." multiple possible nodes to/from which exists an edge. Therefore, an arrow drawn to/from "..." should be seen as multiple edges. In patterns 2 and 3 nodes Ch1 to Ch3 can also be terminal. The compaction of pattern 6 is not implemented.*

one by one and check whether the pattern can be found in the current AND-OR graph. If we find an occurrence, we immediately perform the corresponding compaction before looking for the next pattern. We detect the patterns in the order presented in Section 3. We chose this order because compacting Patterns 1 to 3 first, makes it possible to detect Patterns 4 and 5 in the same iteration. We stop iterating when no pattern is detected. Note that in principle we could stop after any number of iterations.

Implementation. The current implementation is in Prolog. It uses three types of facts to represent nodes and edges. The `single_edge/2` facts represent the edges from an OR or terminal node to an OR node. We refer to these as *single edges*. We do not define an AND node explicitly but use the `and_edge/2` facts to represent an edge from a conjunction (encoded as a list) of two or more OR or terminal nodes to an OR node. We refer to these edges as *and edges*. For *terminal nodes* we use `node/2` facts with the node as a first argument and its probability as second. Our implementation is not optimized in any way. It uses exhaustive search over the AND-OR graph. Appendix B illustrates the algorithms for pattern detection.

Soundness and completeness. Our algorithm is **deterministic** and ensures the same output for the same input. It is **sound** in the sense that the compacted AND-OR graph is equivalent to the initial one in terms of WMC. For Patterns 1 to 3, the soundness argument is based on results from program transformation. Pattern 1 removes a goal that is proven by one fact, Pattern 2 and 3 unfold a goal for which there is one clause in the (ground) program. The soundness of Pattern 4 and 5 is proven in (Mantadelis and Janssens 2010). As described in Section 3 our algorithm does neither detect nor compact at least one pattern (Pattern 6). Thus, the transformed graph is not of minimal size. Therefore, our algorithm is **not complete**.

Complexity. Our implementation allows direct access to the AND-OR graph. It ensures reading, adding or deleting in constant time. That is why, once a pattern is detected the compaction step consumes time linear to the number of edges which are affected. Detecting a pattern is the bottleneck of our algorithm and deserves further analysis.

For an arbitrary AND-OR graph G we denote with N_{or} the number of *single edges*, with N_{and} the number of *and edges* and with n_{term} – the number of terminal nodes.

1. Detecting Pattern 1 requires checking the *single* edges for edges between an OR node A and a terminal node B. There are at most N_{or} such edges. To find out that A only depends on B, we collect the m nodes A depends on and ensure that m equals 1. Since there may exist at most $N_{or} + N_{and}$ edges towards A, $m \leq N_{or} + N_{and}$. Thus, the total complexity bound of detecting Pattern 1 is $O(N_{or}^2 + N_{or} * N_{and})$.
2. For Pattern 2, we consider the N_{and} *and* edges from an AND node C to an OR node B. Next, we check that B depends only on C. As argued for pattern 1, this requires $N_{or} + N_{and}$ operations. For B we require another $N_{or} + N_{and}$ operations to collect all nodes (A) which depend on B. In total we require at most $N_{and} * (N_{or} + N_{and} + N_{or} + N_{and})$ operations with $O(N_{or} * N_{and} + N_{and}^2)$ as complexity bound.
3. Detecting Pattern 3 requires one loop over the *single* edges to find an edge between two OR nodes (A and B), such that (i) B is not terminal and (ii) only A depends

on B. The cost of checking (i) is a constant. Similar to the previous patterns, (ii) requires at most $N_{or} + N_{and}$ checks. Therefore, the complexity of detecting Pattern 3 is $O(N_{or}^2 + N_{or} * N_{and})$.

4. For an AND cluster we need to find the *and* edges with AND nodes B that have at least two terminal nodes. An *and* edge has m nodes dependent on B. We keep the ones which (i) are terminal and (ii) the ones only B depends on. For (i) we need to check the m nodes and keep the m_t terminal ones. For (ii) there are $N_{or} + N_{and}$ checks for every of the m_t nodes. Because $m_t \leq n_{term}$ the total number of operations is at most $N_{and} * ((N_{and} + N_{or}) + n_{term} * (N_{and} + N_{or}))$. The complexity bound for detecting an AND cluster is $O(N_{and}^2 * n_{term} + N_{or} * N_{and} * n_{term})$.
5. OR Cluster I. The analysis of the complexity of detecting an OR cluster I is similar to the one for AND clusters. Detecting OR clusters considers *single* edges. Therefore, complexity bound is, $O(N_{or}^2 * n_{term} + N_{or} * N_{and} * n_{term})$.
6. OR cluster II. This pattern is not implemented in our code. Here we give a theoretical complexity for the simple case depicted in Table 1. To find an OR node A which depends on 2 (B_1 and B_2) AND nodes we check every *and* edge with every other *and* edge. This requires $\frac{N_{and}(N_{and}-1)}{2}$ checks. B_1 and B_2 should (i) depend on a common node C and (ii) have a number of terminal nodes. According to the analysis of the previous patterns, checking the nodes on which B_1 depends needs at most $N_{and} + N_{or}$ operations (same for B_2). To find the common node C we can use set intersections with at most $(N_{and} + N_{or})^2$ operations. Finally, from each of the terminal nodes of B_1 and B_2 we need to check if only one node depends on it. From the analysis of the previous patterns this requires at most $n_{term}(N_{and} + N_{or})$ checks. This gives a total complexity bound of $O(N_{and}^2 * (N_{and} + N_{or})^3 + N_{and}^2 * (N_{and} + N_{or})^2 * n_{term})$.

5 Experiments

For our experiments we used MetaProbLog and ProbLog2. Our compaction algorithm applies after the proof collection step and before the conversion to a Boolean formula.

Our hypothesis is that *using compaction increases the overall performance of the system*. More specific, we claim that the compaction decreases the time and memory consumption for the next steps in the execution pipelines of MetProbLog and ProbLog2.

To support our hypothesis we should answer the following questions:

Q 1: What is the gain (if any) in time and memory consumption due to compaction?
Q 2: What is the compaction cost and in which cases it is unfeasible using it?
Q 3: What is the relationship between compaction and size of the initial AND-OR graph?
Q 4: Does compaction affect MetaProbLog and ProbLog2 differently?

5.1 Experimental setup

We ran our experiments on a 4-core/8-thread Intel® machine with 16GB of memory. We used the data set from (De Raedt et al. 2007). It is a real-world biological dataset of Alzheimer genes which corresponds to a directed probabilistic graph of 11530 edges and 5220 nodes. 20 subgraphs are extracted from the initial graph. They have increasing

sizes: $G_1 \subset G_2 \subset .. \subset G_{20}$, with 200 edges for G_1, 400 – G_2, and so forth. We used graphs G_5 to G_{20} after excluding duplicate edges with the `path` program of Figure 2.a. For each of the graphs we ran 6 `path` queries (Mantadelis 2012, Chapter 6) and (De Raedt et al. 2007), over the genes (nodes) 'hgnc_582', 'hgnc_983' and 'hgnc_620':

q1: ('hgnc_582', 'hgnc_983'), *q2*: ('hgnc_983', 'hgnc_582'), *q3*: ('hgnc_620', 'hgnc_983'),
q4: ('hgnc_983', 'hgnc_620'), *q5*: ('hgnc_582', 'hgnc_620'), *q6*: ('hgnc_620', 'hgnc_582')

For *MetaProbLog*, we used the exact probabilistic inference of the system (using SLG resolution and BDD script generation). Boolean formula generation used all optimizations described in (Mantadelis 2012, Chapter 3). For *ProbLog2*, we selected the options query-based CNF conversion, c2d (Darwiche 2004) for sd-DNNF compiler and 'fileoptimized' evaluation. Inference was timed out after 1 hour.

To measure the complexity of an AND-OR graph we use its size as defined in Definition 2. If $|G_1| = |G_2|$ and $|V_1| < |V_2|$, then G_1 is more densely connected than G_2.

Definition 2
The **size** of a graph $G = (V, E)$ with V the set of nodes and E the set of edges is $|G| = |V| + |E|$.
The **size** of a ProbLog program P is the number of ground probabilistic facts of P.

5.2 Results

MetaProbLog and ProbLog2 produce the same AND-OR graphs and our compaction algorithm applies the same transformations in both cases. In every program a number of Patterns 2 and AND Clusters are detected. E.g. for query 1 and a program of size 3930 there are 1518 *single branches* and 1518 *AND clusters*. We determine the memory gain from the compaction rate $C^+ = (|G_i| - |G_c|)/|G_i|$, where G_i and G_c are the initial and the compacted AND-OR graphs. Table 2 summarizes the compaction rate per query.

Program Size	q1	q2	q3	q4	q5	q6
1722	39.83	40.04	39.86	40.74	32.16	40.04
2056	39.56	39.75	39.56	40.70	31.28	39.75
2384	39.86	40.03	39.86	40.41	32.45	40.03
2700	39.49	39.67	39.49	39.98	32.50	39.67
3018	39.63	39.79	39.63	40.49	33.54	39.79
3334	39.25	39.43	39.25	39.67	34.36	39.43
3620	39.35	39.52	39.35	39.60	34.84	39.52
3930	39.50	39.66	39.50	40.14	35.49	39.66
Average	39.56	39.74	39.57	40.21	33.33	39.74

Table 2. *Compaction rate (C^+) in %.*

Table 2 shows almost constant compaction rate with average around 39%. It is because on the one hand, increase of the program size results in bigger AND-OR graphs (see Figure 4); on the other hand, larger AND-OR graphs contain more patterns to compact.

The time consumed by the compacting of AND-OR graphs (T_c) is summarized in Figure 4. Although close to linear, if we draw trend lines and analyze the growth tendencies we discover a polynomial dependency on the size of the graph. This growth rate is consistent with the complexity bounds given in Section 4.

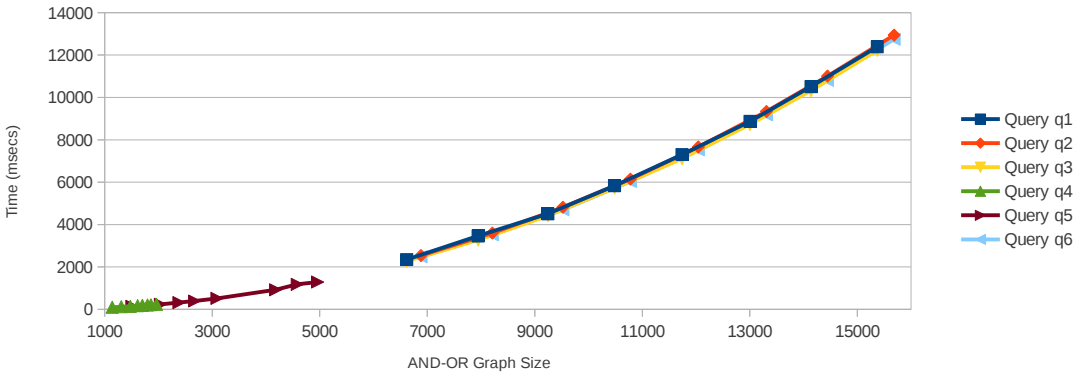


Fig. 4. Time consumption of AND-OR Graph compaction per Graph Size.

To judge the time gain due to AND-OR graph compaction we measured the execution time of each step in the inference pipeline. Of major importance are the Boolean formula conversion time (T_{bc}), its compilation to ROBDD or sd-DNNF (T_{kc}) and its evaluation (T_e). The total (processing-compilation) time $T_{pc} = T_{bc} + T_{kc} + T_e$ does not include the compaction time T_c. We ran every query twice: once with and once without compaction and collected their process-compilation times T_{pc}^{C} and T_{pc}^{NoC} correspondingly. Table 3 summarizes our timing results.

Given the time gain ratio $T^{+} = (T_{pc}^{NoC} - T_{pc}^{C})/T_{pc}^{NoC}$ from Table 3 we deduce: *first*, a decrease in the total processing-compilation time in 86% of the cases which terminated. In the best case we perform more than 65% faster, while in the worst case – 80% slower. Our compaction does not depend on the next steps which use heuristics. Such cases (like query $q5$ on program 5 – of size 3018, both systems perform bad) show that it is not always the graph size which determines the performance of the next steps. *Second*, there are 4 fewer time-outs and 1 time-out more, due to compaction. Although this is a small number it still proves that our compaction method enables ProbLog to deal with larger problems. *Third*, although the time gain is different for MetaProbLog and ProbLog2 it is obvious that both systems benefit from the compaction. This supports the claim that our approach applies equally well to different probabilistic inference methods.

Tables 4 and 5 show the difference between T^{+} and T_c for MetaProbLog and ProbLog2 respectively. Negative values show that compaction costs more than what we gain. For MetaProbLog this is in 73% of the cases, that is, in 73% of the cases using the compaction is infeasible. For ProbLog2 it is the other way around: it doesn't benefit from the compaction only in 25%. We also notice that in most of the cases with increase of the program size we observe that the gain becomes higher than the cost (positive values in the tables). Our algorithm is not optimized. Therefore, times can be further improved.

6 Conclusions, Future and Related Work

Our paper presents a method for compacting AND-OR graphs in the context of probabilistic inference. In the first step, a query q is proven (by SLG resolution or a similar method). The following steps convert the collected proofs into a suitable representation

	q1		q2		q3	
Program Size	MetaProbLog	ProbLog2	MetaProbLog	ProbLog2	MetaProbLog	ProbLog2
1722	34.39	52.64	27.08	56.39	40.00	66.45
2056	52.93	37.36	12.08	48.36	54.45	57.15
2384	25.98	38.44	45.37	51.52	48.62	53.11
2700	-	0.27	-34.61	55.82	57.56	39.42
3018	-	-	no time-out	time out	57.10	34.08
3334	-	-	-	-	no time-out	no time-out
3620	-	-	-	-	61.00	-
3930	-	-	-	-	-	-
Average:	37.77	32.18	12.48	53.02	53.12	50.04

	q4		q5		q6	
Program Size	MetaProbLog	ProbLog2	MetaProbLog	ProbLog2	MetaProbLog	ProbLog2
1722	12.50	51.91	33.33	16.33	26.57	57.17
2056	17.65	57.44	40.76	15.92	10.52	47.75
2384	27.78	38.93	32.74	16.74	45.15	50.61
2700	5.88	56.15	-27.62	8.38	-38.90	30.99
3018	15.00	11.60	-80.40	-9.30	no time-out	19.40
3334	5.56	11.95	-23.78	-17.81	-	-
3620	15.00	-48.26	-	-	-	-
3930	-5.56	18.01	-	-	-	-
Average:	11.73	24.72	-4.16	5.04	10.84	41.18

Table 3. *Time gain ratio (T^+) in % after AND-OR graph compaction. "no time-out" indicates that due to compaction the system didn't time out in* 1 *hour. "time-out" indicates that with compaction the system timed out in* 1 *hour, while without compaction – it didn't. "-" indicates a time out in the two cases – with and without compaction.*

Program Size	q1	q2	q3	q4	q5	q6
1722	-1854	-2396	-2216	-69	-108	-2404
2056	28588	-3174	-2962	-90	-143	-3211
2384	9412	-2644	-4148	-112	-250	-2671
2700	-	-34722	-2076	-139	-3898	-37499
3018	-	no time-out	32038	-151	-47702	no time-out
3334	-	-	no time-out	-172	-27111	-
3620	-	-	214186	-182	23053	-
3930	-	-	-	-206	54215	-

Table 4. *Difference between time gain ($T_{pc}^{NoC} - T_{pc}^{C}$) and compaction time ($T_c$) in msecs. MetaProbLog.*

which allows fast result evaluation. Our work aims at decreasing the size of the collected proofs by detecting and compacting subgraph patterns in an AND-OR graph.

Our algorithm first generates an AND-OR graph from the collected proofs. Then it detects one pattern at a time and transforms it appropriately. We decrease the size of the AND-OR graph and reduce the execution time of the next steps. The experimental results (summarized in Section 5) show time gain and compression rates which prove that our compaction indeed improves the performance of probabilistic inference algorithms.

It is important to note that our patterns are defined without making any assumptions

Program Size	q1	q2	q3	q4	q5	q6
1722	1343	1128	1006	9	39	1256
2056	6386	2106	2127	21	99	2042
2384	13414	3275	3606	-20	107	3071
2700	-2642	28321	21241	42	210	12003
3018	-	time-out	165751	-164	-5652	501178
3334	-	-	no time-out	-179	-74311	-
3620	-	-	-	-499	-	-
3930	-	-	-	-165	-	-

Table 5. *Difference between time gain ($T_{pc}^{NoC} - T_{pc}^{C}$) and compaction time ($T_c$) in msecs. ProbLog2.*

about the methods used in the next steps and that our compaction is independent of the hosting PLP system. Both MetaProbLog and ProbLog2 benefit from the compaction.

To get the best of our approach in the future we aim at improving its implementation and include detection and compaction of more patterns: eg., detecting and compacting some forms of Pattern 6.

Our method is comparable to several other approaches. We experimentally compared it to the complete algorithm (Mantadelis and Janssens 2010) and (Mantadelis 2012, Chapter 5) (see Appendix C), and obtained similar efficiency gains, despite the fact that our algorithm is incomplete. Related work such as (Gogate and Domingos 2010; Hintsanen 2007; Dechter and Mateescu 2007) also exploits the model structure in order to improve the efficiency of the inference algorithm. The approach presented in (Gogate and Domingos 2010) simplifies the model by unit propagation, resolution and subsumption elimination, and by decomposition. We can see our Patterns 1 to 3 as a case of unit propagation and elimination. The work presented in (Hintsanen 2007) uses series-parallel subgraphs for which they compute the probability polynomially. These particular case of subgraphs resembles our AND-/OR- clusters. Search on AND-OR graphs in the framework of probabilistic inference is discussed in (Dechter and Mateescu 2007). They exploit subgraph independencies to reduce the size of the search space. Our AND-/OR- clusters represent also independent parts.

Our approach can be seen as a form of lifting (Poole 2011). Lifting approaches exploit general model properties like symmetry (e.g.(Kersting et al. 2009)). Our method exploits graph properties in more specific way (i.e. specific patterns) and our compaction preserves the WMC.

References

Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput. 35,* 8 (Aug.), 677–691.

Chen, W. and Warren, D. S. 1996. Tabled evaluation with delaying for general logic programs. *Journal of the ACM 43*, 43–1.

Darwiche, A. 2004. New advances in compiling CNF into decomposable negation normal form. In *Proceedings of the 16th European Conference on Artificial Intelligence.* 328–332.

Darwiche, A. 2009. *Modeling and Reasoning with Bayesian Networks.* Cambridge University Press. Chapter 12.

Darwiche, A. and Marquis, P. 2002. A knowledge compilation map. *Journal of Artificial Intelligence Research 17*, 229–264.

De Raedt, L., Kimmig, A., and Toivonen, H. 2007. Problog: a probabilistic prolog and its application in link discovery. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*. AAAI Press, 2468–2473.

Dechter, R. and Mateescu, R. 2007. And/or search spaces for graphical models. *Artificial Intelligence 171,* 2-3, 73–106.

Fierens, D., Broek, G. V. D., Renkens, J., Shterionov, D., Gutmann, B., Thon, I., Janssens, G., and de Raedt, L. 2013. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming, Special Issue on Probability, Logic and Learning*.

Fierens, D., Van den Broeck, G., Thon, I., Gutmann, B., and De Raedt, L. 2011. Inference in probabilistic logic programs using weighted cnf's. In *UAI*. 211–220.

Gogate, V. and Domingos, P. 2010. Formula-based probabilistic inference. In *UAI 2010, Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*. AUAI Press, 210–219.

Hintsanen, P. 2007. The most reliable subgraph problem. In *PKDD 2007: Proceedings of the 11th European conference on Principles and Practice of Knowledge Discovery in Databases*. Springer-Verlag, Berlin, Heidelneg, 471–478.

Janhunen, T. 2004. Representing normal programs with clauses. In *Proceedings of the 16th European Conference on Artificial Intelligence*. IOS Press, 358–362.

Kersting, K., Ahmadi, B., and Natarajan, S. 2009. Counting belief propagation. In *Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence*. UAI '09. AUAI Press, Arlington, Virginia, United States, 277–284.

Mantadelis, T. 2012. Efficient algorithms for prolog based probabilistic logic programming. Ph.D. thesis, Arenberg Doctoral School of Science, Engineering & Technology, KULeuven.

Mantadelis, T. and Janssens, G. 2010. Variable compression in problog. In *Proceedings of the 17th international conference on Logic for programming, artificial intelligence, and reasoning*. LPAR'10. Springer-Verlag, Berlin, Heidelberg, 504–518.

Muise, C., Mcilraith, S. A., Beck, J. C., and Hsu, E. 2012. DSHARP: Fast d-DNNF compilation with sharpSAT. In *Canadian Conference on Artificial Intelligence*. Canadian Conference on Artificial Intelligence.

Poole, D. 2011. Logic, probability and computation: Foundations and issues of statistical relational ai. In *LPNMR*, J. P. Delgrande and W. Faber, Eds. Lecture Notes in Computer Science, vol. 6645. Springer, 1–9.

Sato, T. and Kameya, Y. 1997. Prism: a language for symbolic-statistical modeling. In *In Proceedings of the 15th International Joint Conference on Artificial Intelligence*. 1330–1335.

Somenzi, F. 2005. Cudd: Colorado university decision diagram package, programmer's manual. http://vlsi.colorado.edu/ fabio/cudd/.

Appendix A The Compaction steps for our Running Example

This section depicts the compaction steps for the AND-OR graph of Figure 2.c. Each intermediate graph is derived after applying one or more transformations on the previous one.

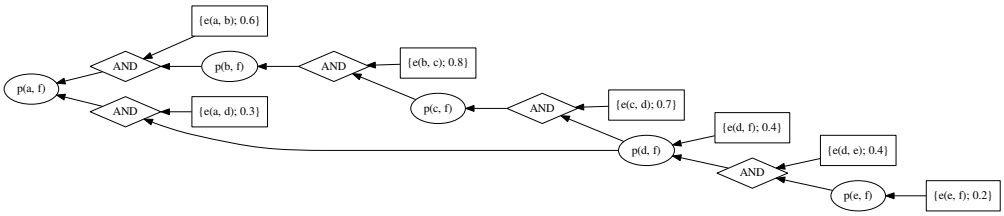


Fig. A 1. Initial AND-OR graph.

On the initial graph (Figure A 1) Patterns 1 and 2 are detected and compacted. The resulting graph is shown in Figure A 2.

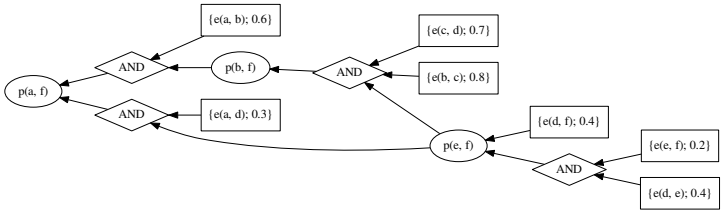


Fig. A 2. First iteration: compaction by Patterns 1 and 2.

.

Then, our algorithm detects two *AND* clusters. The result after compaction is shown in Figure A 3.

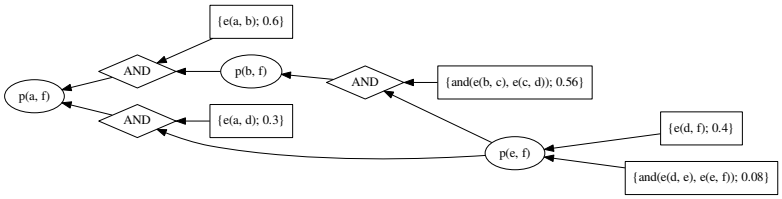


Fig. A 3. First iteration: compaction of two *AND Clusters*.

Figure A 4 depicts the AND-OR graph after detecting one *OR* cluster and applying the corresponding transformation.

As the first iteration performed some compactions, we start the second iteration. The second iteration detects a Pattern 1 and a Pattern 2. The result is in Figure A 5.

The next compaction is after detecting one *AND* cluster. The algorithm will check for *OR* clusters and not detect any. After this compaction there are no more patterns to detect. The algorithm, though, will do one more iteration. This is required since after a compaction, new patterns may appear. In total we need 3 iterations. The final compacted AND-OR graph is in Figure A 6.

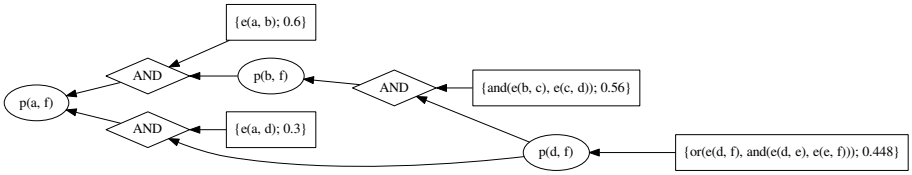


Fig. A 4. First iteration: compaction of an *OR Cluster*.

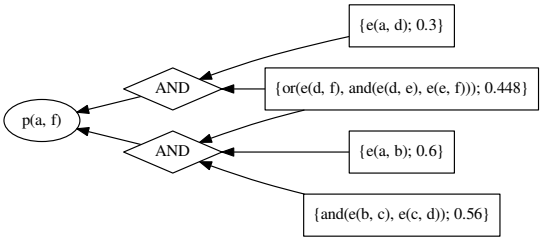


Fig. A 5. Second iteration: compaction by Patterns 1 and 2.

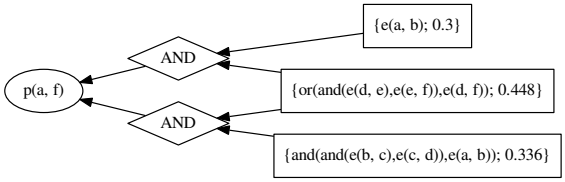


Fig. A 6. The final compacted AND-OR graph.

Appendix B Algorithm Code

Here we presents our algorithm for pattern detection. The AND-OR graph is represented by two types of edges *single edges* and *and edges*. The corresponding Prolog facts are `single_edge/2` and `and_edge/2`. Terminal nodes are encoded as `node/2` facts. A *single* edge has as first argument an OR node and as second argument an OR or a terminal node on which it depends. An *and* edge has also an OR node as a first argument but the second argument is a list of OR or terminal nodes which participate in a conjunction. Terminal nodes have as first argument the node itself and the node's probability as second argument.

Algorithm 1 illustrates how we detect the *simple* patterns (1, 2 and 3). Algorithm 2 detects *complex* patterns (4 and 5, AND Cluster and OR Cluster I, respectively).

Algorithm 1 Algorithms for Detecting Patterns 1 to 3.

Pattern 1

```
detect_pattern_1(ORNode, TermNode) :-
  single_edge(ORNode, TermNode),
  findall(CollectTermNode, (node(CollectTermNode, _Prob),
          single_edge(ORNode, CollectTermNode)), CollectedTermNodes),
  CollectedTermNodes = [TermNode],
  \+ and_edge(ORNode), _ANDNode),
  \+ or_edge(ORNode), _ORNode2).
```

Pattern 2

```
detect_pattern_2(ORNode, ANDNode) :-
  and_edge(ORNode, ANDNode),
  \+ single_edge(ORNode, _ORNode2),
  findall(CollectANDNode, and_edge(ORNode, CollectANDNode), CollectANDNodes),
  CollectANDNodes == [ANDNode],
  findall(ParentOR, single_edge(ParentOR, ORNode), ParentORs),
  findall(ParentOR, (and_edge(ParentAND, ChildList),
           member(ORNode, ChildList, ParentANDs),
  (ParentORs \== []; ParentANDs \== []).
```

Pattern 3

```
detect_pattern_3(FatherORNode, ChildORNode) :-
  single_edge(FatherORNode, ChildORNode),
  (single_edge(ChildORNode, _ORNode2); and_edge(ChildORNode, _ANDNode)),
  findall(ParentOR, single_edge(ParentOR, ChildORNode), ParrentORs),
  ParrentORs == [FatherORNode],
  \+ (and_edge(ParentAND, ChildList), member(ChildORNode, ChildList)).
```

To detect patterns 4 and 5 (AND- and OR- clusters) we need to exclude from the set of child nodes that form the cluster, non-terminal nodes and nodes that have more than one parent.

For a specific AND or OR node A a terminal node which has other parent besides A cannot participate in an AND Cluster or OR Cluster I. The *refinement* steps (`refine_and_cluster_from_or_nodes/2`, `refine_and_cluster_from_and_nodes/2`,

Algorithm 2 Algrorithms for Detecting Patterns 4 and 5.

Pattern 4

```
detect_pattern_4(ANDCluster) :-
  and_edge(ORNode, ANDNode),
  findall(CollectTermNode, (node(CollectTermNode, _Prob),
          member(CollectTermNode, ANDNode)),
          ANDCluster1),
  refine_and_cluster_from_or_nodes(ANDCluster1, ANDCluster2),
  refine_and_cluster_from_and_nodes(ANDCluster2, ANDCluster).
```

Pattern 5

```
detect_pattern_5(ORCluster) :-
  single_edge(ORNode, _TermNode),
  findall(CollectTermNode, (node(CollectTermNode, _Prob),
          single_edge(ORNode), CollectTermNode)),
          ORCluster1),
  refine_or_cluster_from_and_nodes(ORCluster1, ORCluster2),
  refine_or_cluster_from_or_nodes(ORCluster2, ORCluster).
```

`refine_or_cluster_from_and_nodes/2`, `refine_or_cluster_from_or_nodes/2`) in detecting an AND Cluster and an OR Cluster I remove such terminal nodes. Due to space limitations, here we do not show the implementation of neither of the refinement predicates.

Appendix C Comparison to existing variable compression in ProbLog

The current work was inspired by the variable compression method described in (Mantadelis and Janssens 2010). Here we compare the properties and performance of the two methods – the *variable compression* and the *AND-OR graph compaction*.

System: *Variable Compression* is incorporated in MetaProbLog. *AND-OR Graph Compaction* works in both MetaProbLog and ProbLog2. The *AND-OR Graph Compaction* algorithm does not dependent on the hosting system, every PL inference mechanism can use it.

Data format: Both *Variable Compression* and *AND-OR Graph Compaction* are applied on a set of proofs. While the *Variable Compression* method can be applied on a Boolean formula in a Normal Form – DNF or CNF, the *AND-OR Graph Compaction* uses an intermediate representation of the collected proofs, namely the AND-OR graph. AND-OR graphs are more general than CNF or DNF and we can extract a DNF or a CNF from them.

Completeness: The *Variable Compression* algorithm is complete, while *AND-OR Graph Compaction* is not complete.

Implementation: Both methods are implemented as Prolog programs. The *Variable Compression* implementation is optimized by using bit strings to represent a proof. Each bit string contains one bit for every fact. If the i^{th} bit is 1 then the i^{th} fact participates in the proof. On the other hand, the current implementation of the *AND-OR Graph Compaction* is an exhaustive search over the graph with no optimizations.

Negated literals support: The *Variable Compression* does not support negated literals in contrast to the *AND-OR Graph Compaction*.

Tabling: To generate the AND-OR graph we use the nested trie representation of the collected proofs (MetaProbLog). On the other hand *Variable Compression* applies on the flat DNF.

To compare the performance of the two methods we used queries $q4$ and $q6$ (see Section 5.1). Inference on query $q4$ is easy – the number of derived proofs and involved probabilistic facts is small. Query $q6$, on the other hand, is hard. Our tests show the performance of the two algorithms on these extreme cases. Table C 1 and table C 2 summarize the experimental results with respect to size and time consumption.

	Query q4		Query q6	
Program Size	AND-OR Graph	Var Compression	AND-OR Graph	Var Compression
1722	40.70	80.00	40.03	50.00
2056	40.67	80.00	39.75	38.39
2384	40.38	42.86	40.03	37.70
2700	39.95	42.86	39.67	39.77
3018	40.49	42.86	39.79	37.06
3334	39.64	42.86	39.42	34.02
3620	39.57	42.86	39.51	33.88
3930	40.12	57.14	39.65	33.74
Average	40.19	53.93	39.73	38.07

Table C 1. *Compaction ratio in % for queries $q4$ and $q6$.*

To compare the compression rate of the two methods we have used a slightly different

measurement from the one of Section 5.2. We count the logical variables in the DNF before and after the compression in the case of *Variable Compression*, and the variables in the nested tries. While the DNF representation includes only probabilistic facts as variables, the nested tries include also references to ground goals. Although, these counts are not comparable, their rates are as they both represent the size difference between initial and compressed data.

Table C 1 shows that *Variable Compression* has better compression rates in the case of query $q4$. For the harder case of query $q6$ the gain rates due to either of the approaches are very similar (around 39%).

	Program Size	Variable Compression	AND-OR Graph MetaProbLog	AND-OR Graph ProbLog2
1.	1722	11.11	26.57	57.17
2.	2056	29.77	10.52	47.75
3.	2384	36.14	45.15	50.61
4.	2700	14.31	no gain	30.99
5.	3018	14.44	no time-out	19.40
6.	3334	29.10	no gain	no gain
	Average			
	1. - 3.	25.67	27.41	51.84
	1. - 5.	21.15	n/a	41.18
	1. - 6.	22.93	n/a	n/a

Table C 2. *Time gain ratio in % for query* $q6$.

The timing result for the *AND-OR Graph compaction* method are in Section 5.2. The total inference time for query $q4$ is very small with and without *Variable Compression* and does not allow to draw reliable conclusions. That is why we present only the inference times for query $q6$.

It is obvious from Table C 2 that the time gain in the case of the *Variable Compression* method is more moderate and does not reach high values. The three different average values are over different ranges of program sizes: 1. - 3. ranges over the programs with size 1722 to 2384; 1. - 5. from program size 1722 to 3018; 1. - 6. from program size 1722 to 3334. On average our algorithm results in higher time gains.