Compiling Probabilistic Logic Programs into Sentential Decision Diagrams

Jonas Vlasselaer, Joris Renkens, Guy Van den Broeck, and Luc De Raedt

Department of Computer Science, KU Leuven, Belgium
firstname.lastname@cs.kuleuven.be

Abstract. Knowledge compilation algorithms transform a probabilistic logic program into a circuit representation that permits efficient probability computation. Knowledge compilation underlies algorithms for exact probabilistic inference and parameter learning in several languages, including ProbLog, PRISM, and LPADs. Developing such algorithms involves a choice, of which circuit language to target, and which compilation algorithm to use. Historically, Binary Decision Diagrams (BDDs) have been a popular target language, whereas recently, deterministic-Decomposable Negation Normal Form (d-DNNF) circuits were shown to outperform BDDs on these tasks. We investigate the use of a new language, called Sentential Decision Diagrams (SDDs), for inference in probabilistic logic programs. SDDs combine desirable properties of BDDs and d-DNNFs. Like BDDs, they support bottom-up compilation and circuit minimization, yet they are a more general and flexible representation. Our preliminary experiments show that compilation to SDD yields smaller circuits and more scalable inference, outperforming the state of the art in ProbLog inference.

1 Introduction

The interest in combining probabilistic reasoning and logic has grown strongly in recent years and led to the development of different software packages, e.g. PRISM [1], PITA [2] and ProbLog [3, 4]. All of them offer a probabilistic programming language, possibly with some restrictions, together with a variety of exact and approximate algorithms for inference and learning. Although specifically designed to support different models expressed by the language, the inference algorithms are often reduced to the well-studied task of weighted model counting [5–7]. This allows one to solve the probabilistic inference task using state-of-the-art solvers introduced by other communities.

To reduce the probabilistic inference task to weighted model counting, one has to perform two steps. First, the probabilistic logic program is converted into a propositional formula which contains all possible worlds relevant to answering the queries with the evidence. This formula is the relevant grounded logic program without cycles or its equivalent Boolean circuit [7]. Second, this formula is compiled into a more tractable target representation that supports weighted model counting in an efficient way. This is a well-studied task known

as knowledge compilation [8]. The most general language known to support efficient weighted model counting is d-DNNF (deterministic-Decomposable Negation Normal Form).

A knowledge compilation approach to probabilistic inference raises two challenges which, as shown later, are closely related: (1) how to choose a suitable target representation, and (2) how to compile a Boolean circuit. For example, historically OBDDs (Ordered Binary Decision Diagrams) [9], which are a subset of d-DNNF, where often used as target representation [3]. Recent work, however, showed that the use of more general d-DNNFs allows one to compile more complex programs [7]. The choice of target representation has a direct impact on the scalability of exact probabilistic inference in probabilistic logic programs.

In this paper, we propose to use of the recently introduced Sentential Decision Diagrams (SDDs) [10] as target language. Unlike d-DNNFs, an efficient `apply` operator is available for (uncompressed) SDDs [11], which allows one to efficiently conjoin and disjoin two SDDs. This operator can be used for a bottom-up compilation approach, as also supported by OBDDs. SDD compilation was recently shown to outperform d-DNNF compilation for inference in probabilistic graphical models [12]. This paper investigates whether using SDDs instead of d-DNNFs yields similar benefits for reasoning with probabilistic logic programs. Preliminary experiments show the promise of the technique.

2 Background and Related Work

In this section, we illustrate the pipeline of probabilistic logic program inference, from a ProbLog program to a weighted model counting problem on a general Boolean circuit, to an OBDD or d-DNNF circuit that permits efficient inference. We refer to [4] for a detailed description of the pipeline.

2.1 Probabilistic Logic Programs

A probabilistic logic program is a logic program in which some of the facts are annotated with probabilities. The small program shown below models a *social network* with a domain of three persons which all possibly smoke. The goal of the program is to compute the probability for each person that they actually smoke, based on their stress-level and friends.

```
0.4 :: friends(a, b).        0.1 :: stress(a).
0.5 :: friends(b, a).        0.5 :: stress(b).
0.8 :: friends(a, c).        0.9 :: stress(c).
0.9 :: friends(c, a).
0.2 :: friends(c, b).        smokes(X) :- stress(X).
0.1 :: friends(b, c).        smokes(X) :- friends(X, Y), smokes(Y).
```

Before this theory can be compiled, the program needs to be grounded as shown below. We dropped the probabilistic facts in the grounded program as they are the same as in the non-grounded program.

```
smokes(a) :- stress(a).
smokes(b) :- stress(b).                    smokes(a) :- friends(a, b), smokes(b).
smokes(c) :- stress(c).                    smokes(a) :- friends(a, c), smokes(c).

smokes(b) :- friends(b, a), smokes(a).     smokes(c) :- friends(c, a), smokes(a).
smokes(b) :- friends(b, c), smokes(c).     smokes(c) :- friends(c, b), smokes(b).
```

The grounded program clearly contain cycles, for example `smokes`(a) depends on `smokes`(b) which again depends on `smokes`(a). These cycles need to be broken in order to obtain an Boolean formula which is equivalent to the logic program [13]. This requires the introduction of auxiliary variables, as shown below in the cycle-free ground program for our example. All `smokes-*` atoms are auxiliary variables necessary to break the loops.

```
   smokes(a) :- stress(a).        smokes(a) :- friends(a, b), smokes-a(b).
   smokes(b) :- stress(b).        smokes(a) :- friends(a, c), smokes-a(c).
   smokes(c) :- stress(c).        smokes(b) :- friends(b, a), smokes-b(a).
                                  smokes(b) :- friends(b, c), smokes-b(c).
 smokes-a(b) :- stress(b).        smokes(c) :- friends(c, a), smokes-c(a).
 smokes-a(c) :- stress(c).        smokes(c) :- friends(c, b), smokes-c(b).

 smokes-b(a) :- stress(a).      smokes-a(b) :- friends(b, c), smokes-ab(c).
 smokes-b(c) :- stress(c).      smokes-a(c) :- friends(c, b), smokes-ac(b).

 smokes-c(a) :- stress(c).      smokes-b(a) :- friends(a, c), smokes-ab(c).
 smokes-c(b) :- stress(b).      smokes-b(c) :- friends(c, a), smokes-bc(a).

smokes-bc(a) :- stress(a).      smokes-c(a) :- friends(a, b), smokes-ac(b).
smokes-ac(b) :- stress(b).      smokes-c(b) :- friends(b, a), smokes-bc(a).
smokes-ab(c) :- stress(c).
```

Once a loop-free ground program is obtained, the theory can be easily represented as a Boolean circuit. Figure 1 depicts the Boolean circuit for our example. It contains a leaf variable for every probabilistic fact, and has a root node for every ground atom we want to compute the probability of. Intermediate nodes correspond to individual clauses and auxiliary variables in the loop-free ground program. This circuit can be compiled into a specific target representation for inference (possibly after first encoding it into a CNF), as we discuss next.

2.2 Properties of Target Representations

A number of distinct compilation algorithms exist. Not only their compilation technique differs, but also the language of the circuit they produce. The knowledge compilation map [8] defines a wide range of target languages and extensively

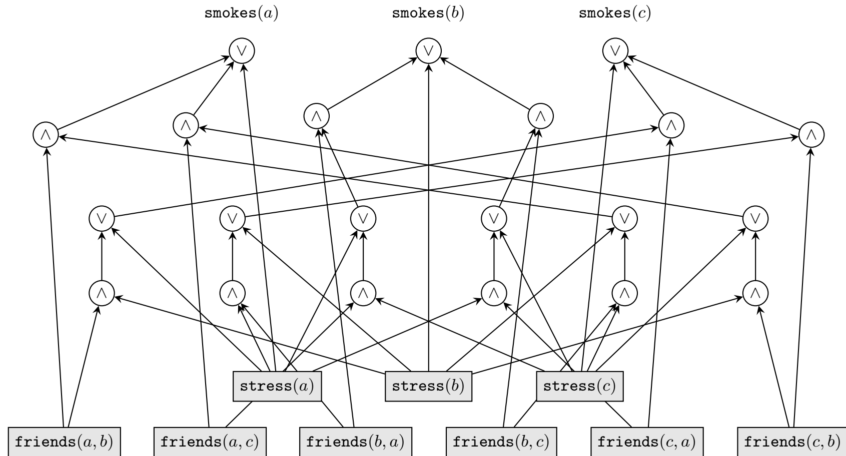


Fig. 1. Boolean circuit for the smokes program.

describes their properties. In this paper, we limit our discussion to the relevant properties of the techniques and languages used for compiling probabilistic logic programs, that is OBDD, d-DNNF and SDD.

In general, one defines three key properties for each target language: its succinctness, the class of tractable transformations it admits, and the class of tractable queries it supports. Succinctness refers to the size of the smallest compiled circuit for every Boolean formula [8]. The succinctness ordering for the languages we consider in this paper is

$$\text{d-DNNF} < \text{SDD} \leq \text{OBDD},$$

where d-DNNF $<$ SDD denotes that d-DNNF is strictly more succinct than SDD, and SDD $\leq$ OBDD denotes that SDD is at least as succinct as OBDD. Intuitively, there exists a Boolean formula whose smallest SDD representation is exponentially larger than its smallest d-DNNF representation, but the smallest OBDD for any formula is at least as big as its smallest SDD. It is an open problem whether SDD $<$ OBDD, although some evidence suggests it is the case [14].

A comparison of the tractable transformations and queries for the languages is summarized in Table 1. The polytime transformations we consider are negation, conjunction and disjunction as they are essential to simplifying the compilation process. The only polytime query we consider is (weighted) model counting as this suffices for probabilistic logic program inference. All languages we consider allow (weighted) model counting in time linear in the size of the obtained circuit. Therefore we do not include this property in further discussion when we compare different compilation techniques or languages.

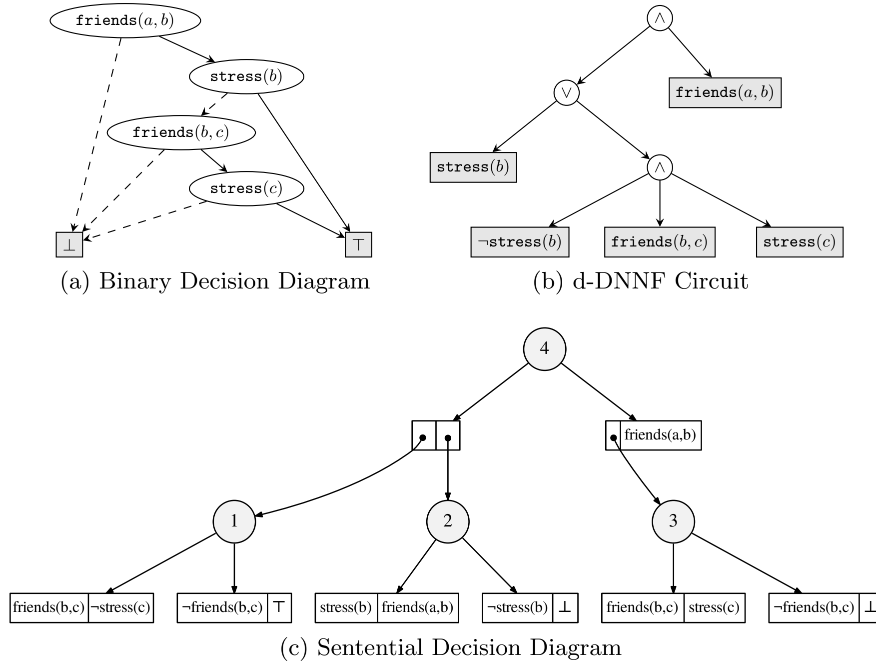


Fig. 2. Efficient circuits for the left-most child of `smokes`(a) in Figure 1, representing the clause `smokes`(a) `:-` `friends`(a, b), `smokes-a`(b) and its subgoals.

2.3 Compiling into OBDD

The first compilation approaches for inference in probabilistic logic programs used OBDDs, a subset of the d-DNNF language, as target representation [3]. Figure 2a depicts an OBDD for the left-most child of `smokes`(a) in Figure 1. It encodes the Boolean function for the clause `smokes`(a) `:-` `friends`(a, b), `smokes-a`(b) and its subgoals. A circular node represent a decision, whether the variable in its label is true or false. Outgoing solid edges denote the variable being true, and dashed edges denote the variable being false. When a terminal is reached, the function is determined to either be true ($\top$) or false ($\bot$).

The advantage of OBDDs is twofold: (1) they support Boolean combinations (conjunction, etc.) in polytime and, as such, allow a bottom-up compilation approach, and (2) they support minimization of the circuit size, by modifying the variable order that governs the OBDD structure.

Consider the cycle-free grounded logic program of the example introduced before. A bottom-up approach would, for example, first compile `smokes-ab`(c), `friends`(b, c) and `stress`(b), all into different OBDDs. Next, it takes the conjunction of the OBDDs of the former two and disjoins it with the OBDD of the latter. The results is an OBDD which represents `smokes-a`(b). In other words, it compiles the nodes in Figure 1 bottom-up, combining the OBDDs of the

Language	Negation	Conjunction	Disjunction	Model Counting
d-DNNF	?	$\circ$	$\circ$	$\surd$
`SDD`	$\surd$	$\surd$	$\surd$	$\surd$
OBDD	$\surd$	$\surd$	$\surd$	$\surd$

Table 1. Efficient circuit operations for the different languages. ? means "unknown", $\surd$ means "satisfies", while $\circ$ means "does not satisfy unless `P=NP`. The transformations assume a bounded number of operands, that are OBDDs with the same variable order, or SDDs with the same variable tree. See [8] and [10] for more details.

children to obtain the OBDD of their parent. This incremental procedure is continued until the complete program is compiled. To avoid an unexpected growth of the circuit after a conjunction or disjunction of multiple sub-circuits, OBDD compilers support a dynamic variable reordering to minimize the circuit size. A disadvantage of OBDDs, however, is that they are less succinct and possibly exponentially larger in size compared to the smallest possible d-DNNF.

2.4 Compiling into d-DNNF

State-of-the-art inference algorithms for probabilistic logic programs first encode the Boolean circuit as a CNF and next compile this CNF directly into a d-DNNF circuit. This approach compared favorably against bottom-up compilation using OBDDs [7]. Figure 2b depicts a d-DNNF circuit for our example Boolean circuit. A d-DNNF is itself a Boolean circuit, but with certain additional restrictions: negation can only appear in the leafs, the children of a conjunction range over disjoint sets of variables, and the children of a disjunction are mutually exclusive.

The advantage of the d-DNNF language is that it comes with size upper bounds based on treewidth. These are tighter than the size upper bounds based on pathwidth for OBDDs. As such, the complexity bound for d-DNNFs is the same as for probabilistic inference algorithms typically used in other communities, for example probabilistic graphical models. The disadvantage of this language, however, is twofold: (1) d-DNNF compilers require CNF input, and the intermediate encoding in CNF requires the introduction of a set of auxiliary variables, and (2) d-DNNFs do not support circuit minimization.

A cycle-free grounded logic program, as introduced before, can be directly converted into a CNF by taking *Clark's completion* of the rules. This conversion, however, introduces a new auxiliary variable for each rule which has a body with more than one literal. Although these extra variables avoid an exponential blow-up of the CNF, their presence complicates the compilation process significantly. Furthermore, compiling probabilistic logic programs into d-DNNF often leads to circuits that are bigger than necessary because minimization is not supported.

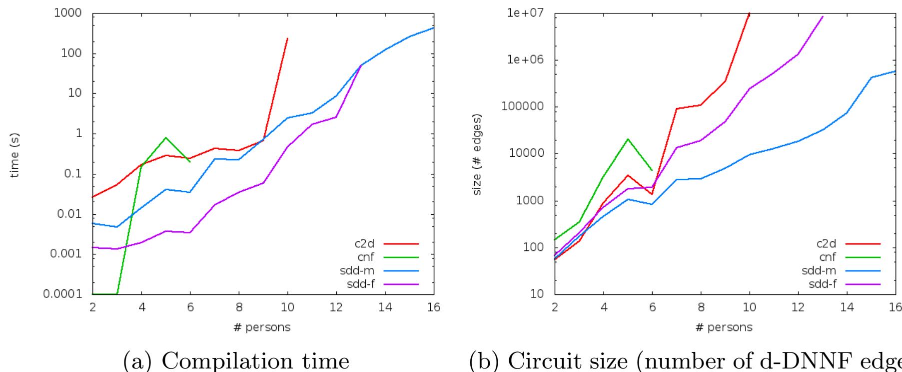
3 Compiling into Sentential Decision Diagrams

The Sentential Decision Diagram (SDD) [10] is a newly introduced target representation which is a strict subset of d-DNNF. Figure 2c depicts an SDD for our example circuit. Circular nodes represent disjunctions and pairs of boxes represent a conjunctions between their two children. More intuitively, circular nodes again represent decisions, but now the decisions are over mutually exclusive complex sentences, and the decisions are themselves represented as SDDs. For example, the decision in node 4 is whether $\texttt{friends}(b, c) \wedge \texttt{stress}(c)$ is true (represented by node 3) or false (represented by node 1). In the first case, one proceeds with checking whether $\texttt{friends}(a, b)$ is true. In the second case, one proceeds with checking node 2, which is itself an SDD.

As a subset of d-DNNF, SDDs satisfy stronger properties which allows one, for example, to conjoin and disjoin two SDDs efficiently by means of an `apply` operator. Moreover, the *compressed* SDDs that we use are canonical, which leads to practically efficient bottom-up-compilation when performing a large number of recursive `apply` operations [11]. Similar to OBDDs, SDDs also support an operation that minimizes their size, by modifying the variable tree that governs their structure [15]. On the other hand, SDDs are a superset of OBDDs and come with tighter upper bounds on their size, based on treewidth. Consequently, probabilistic inference with SDDs combines the desirable properties of both the OBDD and d-DNNF languages: (1) a bottom-up compilation approach, (2) minimization and, (3) succinctness and treewidth upper bounds.

In this work, we investigate techniques for compiling probabilistic logic programs into an SDD circuit. On the one hand, an SDD can be compiled directly from the CNF encoding of the Boolean circuit and, as such, can be used to replace d-DNNFs in the approach of [4]. On the other hand, an SDD can be compiled bottom-up directly from the Boolean circuit, similar as with OBDDs. This bypasses the intermediate CNF encoding and consequently there is no need to introduce extra variables. More concretely, for every extra variable we would have in the equivalent CNF, we now have a corresponding intermediate SDD. By means of the `apply` operator, the SDD sub-circuits can be efficiently conjoined and disjoined.

Our work on compiling probabilistic logic programs is similar to recent work on compiling probabilistic graphical models into SDD [12]. There are two key differences. First, our approach makes use of dynamic minimization, i.e. dynamic reordering of the variable tree when the SDD grows beyond expectations, which was not used before for probabilistic reasoning. Second, breaking the cycles and the more complex rules inherent to logic programs typically require the introduction of a large number of auxiliary variables, and the cost of CNF conversion is much higher for probabilistic logic programs than for typical probabilistic graphical models. We therefore expect a bottom-up compilation approach, which does not require the intermediate CNF representation, to have a much larger impact when used for compiling logic programs.



(a) Compilation time

(b) Circuit size (number of d-DNNF edges)

Fig. 3. Smokers experiments for a growing number of people.

4 Experiments

We experimentally compare the impact of different compilation approaches on the compilation time and size of the compiled circuit. The probabilistic logic program we use is a *social network* with the standard 'Smokers' domain, similar to the one used in [7]. We use random power law graphs to generate the network since they are known to resemble social networks. For every domain size (# of persons), we perform 10 runs and report average results.

For our experiments we do not consider OBDDs as target representation, as previous work showed this approach is outperformed by compilation into d-DNNF [7]. We thus compare four different compilation methods that start from the Boolean circuit representation:

- encoding the Boolean circuit as a CNF, and compiling to d-DNNF (`c2d`);
- encoding the Boolean circuit as a CNF, and compiling to SDD (`cnf`);
- compiling directly into SDD with a fixed default variable tree (`sdd_f`);
- compiling directly into SDD with minimization (`sdd_m`).

Both the d-DNNF compiler (c2d) and SDD compiler we used are developed by the Automated Reasoning Group at UCLA.[1] We use ProbLog as probabilistic programming language. The compilation time for all methods is shown in Figure 3a and the size of the obtained representation in Figure 3b. Note that the size of an SDD and d-DNNF cannot directly be compared, as they are defined on a different syntax. However, every SDD can be transformed into a d-DNNF that is at most three times larger. Hence, the reported SDD size is this upper bound.

We clearly observe the bottom-up compilation approach with SDDs, even without minimization, outperforms the compilation approach which requires an intermediate CNF representation. First, the methods `sdd_f` and `sdd_m` allow to compile probabilistic logic programs with a larger domain size. Second, these

[1] Both compilers are publicly available at http://reasoning.cs.ucla.edu.

methods produce circuits which are similar in size or smaller compared to the circuits compiled by the other two methods. Third, the time necessary to compile the theory is similar or less compared to the other two methods. We also notice the impact of the minimization on the bottom-up approach (`sdd_m`). Minimization does not only allow us to compile programs with a larger domain size, but it also produces smaller circuits in general. Minimization does take some additional compilation time on smaller problems.

5 Conclusion

We investigated the use of a new language, called Sentential Decision Diagrams, for compiling probabilistic logic programs. This language supports an efficient `apply` operator, not available for d-DNNFs, which allows a bottom-up compilation approach. Preliminary experiments show that incrementally compiling the Boolean circuit with SDDs compares favorably against compiling the intermediate CNF representation into d-DNNF or SDD. The new approach lets us compile a social network with a bigger domain size and the obtained circuit is smaller compared to other compilation methods, especially when minimization is used. These results ask for more work and experiments to fully investigate the use of SDDs for compiling probabilistic logic programs.

References

1. Sato, T., Kameya, Y.: PRISM: a language for symbolic-statistical modeling. In: In Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97). (1997) 1330–1335
2. Riguzzi, F., Swift, T.: Tabling and answer subsumption for reasoning on logic programs with annotated disjunctions. In: ICLP (Technical Communications). (2010) 162–171
3. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: a probabilistic Prolog and its application in link discovery. In: In Proceedings of 20th International Joint Conference on Artificial Intelligence, AAAI Press (2007) 2468–2473
4. Fierens, D., Van den Broeck, G., Renkens, J., Shterionov, D., Gutmann, B., Thon, I., Janssens, G., De Raedt, L.: Inference and learning in probabilistic logic programs using weighted Boolean formulas. Theory and Practice of Logic Programming (2013)
5. Chavira, M., Darwiche, A., Jaeger, M.: Compiling relational bayesian networks for exact inference. International Journal of Approximate Reasoning **42**(1) (2006) 4–20
6. Chavira, M., Darwiche, A.: On probabilistic inference by weighted model counting. Artif. Intell. **172**(6-7) (2008) 772–799
7. Fierens, D., Van den Broeck, G., Thon, I., Gutmann, B., De Raedt, L.: Inference in Probabilistic Logic Programs using Weighted CNF's. Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence (UAI) (2011)
8. Darwiche, A., Marquis, P.: A knowledge compilation map. Journal of AI Research **17** (2002) 229–264

9. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys (CSUR) **24**(3) (1992) 293–318
10. Darwiche, A.: SDD: A new canonical representation of propositional knowledge bases. In: Proceedings of the Twenty-Second international joint conference on Artificial Intelligence. (2011) 819–826
11. Van den Broeck, G., Darwiche, A.: On the Role of Canonicity in Bottom-up Knowledge Compilation. CoRR **abs/1404.4089** (2014)
12. Choi, A., Kisa, D., Darwiche, A.: Compiling Probabilistic Graphical Models using Sentential Decision Diagrams. In: Proceedings of the 12th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU). (2013) 121–132
13. Janhunen, T.: Representing normal programs with clauses. In: In Proc. of the 16th European Conference on Artificial Intelligence, IOS Press (2004) 358–362
14. Xue, Y., Choi, A., Darwiche, A.: Basing decisions on sentences in decision diagrams. In: Proceedings of the 26th Conference on Artificial Intelligence (AAAI). (2012) 842–849
15. Choi, A., Darwiche, A.: Dynamic minimization of sentential decision diagrams. In: Proceedings of the 27th Conference on Artificial Intelligence (AAAI). (2013)