aspmc: An Algebraic Answer Set Counter

Thomas Eiter[1], Markus Hecher[1] and Rafael Kiesel[1]

[1] *Vienna University of Technology, Favoritenstrasse 9-11, Vienna, 1040, Austria*

Abstract

We report about aspmc, which is the working prototype implementation of recent advances in Algebraic Answer Set Counting (AASC). AASC refers to counting the answer sets of a normal answer set program with weights over a semiring. This includes many problems that have recently received growing attention, among them probabilistic reasoning, parameter learning, and most probable explanation inference for answer set programs. aspmc employs a *treewidth-aware* cycle-breaking to reduce AASC to Algebraic Model Counting (AMC) over a propositional formula with only a slightly increased *treewidth*. This allows us to lift the performance bounds for AMC in terms of treewidth to AASC. The experimental evaluation of aspmc reveals that these bounds are not only of theoretical interest but also allow us to improve upon the efficiency of other exact counters on standard benchmarks. [1]

Keywords

Probabilistic Reasoning, Answer Set Programming, Model Counting, Treewidth, Semiring

1. Introduction

Recently, there has been a rising interest in reasoning problems for Answer Set Programming (ASP) that go beyond the classical consistency problem. For example, LP$^{\text{MLN}}$[2], P-log [3], PrASP [4], PASP [5] and Problog [6] allow for probabilistic reasoning. Weight Constraints and more generally the asprin framework [7] consider preferential reasoning over answer sets. Finally, algebraic Prolog [8] and Weighted LARS [9] capture and generalize both of these ideas in *Algebraic Answer Set Counting (AASC)*, i.e. weighted answer set counting over semirings. While there are many frameworks allowing such specifications, the corresponding choice of solvers is limited: the clingo solver [10] performs answer set counting by enumeration, which becomes infeasible for millions of answer sets. The dynASP2.5 solver [11] focuses on programs of low treewidth. The Problog solver [12] reduces the problem to Algebraic Model Counting (AMC) [13], for which efficient solvers exist [14, 15, 16]. Other solutions exist for approximate probabilistic queries [17] or most probable answer set inference [18], which are outside of our scope as we aim for exact weighted answer set counting.

While Problog does not accept arbitrary ASP programs as input, we still consider its strategy to be the most promising approach. The basic idea, described in detail in [19], is the following. First one breaks the cyclic dependencies in the input program, obtaining a tight program, where

[1]This is a technical summary of the longer version accepted at KR 2021 [1].

PLP 2021

thomas.eiter@tuwien.ac.at (T. Eiter); markus.hecher@tuwien.ac.at (M. Hecher); rafael.kiesel@tuwien.ac.at (R. Kiesel)
0000-0001-6003-6345 (T. Eiter); 0000-0003-0131-6771 (M. Hecher); 0000-0002-8866-3452 (R. Kiesel)

the answer sets are the models of its Clark completion [20]. The Clark completion can then be compiled it into an equivalent d-DNNF/SDD representation, where AMC is possible in linear time [13]. This approach allows for practically relevant instances to be solved [21]. However, the way cycles in the program's positive dependency graph are broken can have a negative effect on the treewidth.

Cycle breaking is well studied in the ASP community. Among others, Hecher's [22] translation from ASP to SAT guarantees that the treewidth of the SAT instance is $\mathcal{O}(k \log(l))$, where k is the original treewidth and l is the minimum of k and the size of the largest strongly connected component of the dependency graph of the program. Unfortunately, this translation does not preserve models bijectively. In contrast, cycle breaking used in Problog [23, 21] and others [24] preserve models without strong treewidth bounds. We address this and provide the following:

- We show that every program of treewidth k can be translated into an acyclic program with treewidth at most $k \cdot \mathrm{cbs}(\mathrm{DEP}(\Pi))$, where $\mathrm{cbs}(\mathrm{DEP}(\Pi))$, is the *component-boosted backdoor size* of the dependency graph of Π; notably, $\mathrm{cbs}(.)$ is a novel parameter on directed graphs combining backdoor sets and decomposability to measure the cyclicity.
- We provide a prototype implementation aspmc that performs AASC by exploiting the constructive proof of the above result. Our experimental results show that when a program has many answer sets, aspmc outperforms Problog, clingo and lp2sat [24].

2. Algebraic Answer Set Counting

We assume familiarity with the basics of Answer Set Programming (ASP) [25] and use $\mathcal{A}(\Pi)$ and $\mathcal{AS}(\Pi)$ for the atoms and answer sets of a program Π, respectively.

We introduce AASC by using the following minimal instantiation Π_{sm} of the smokers program, which is a standard example from probabilistic logic programming [6].

$$\begin{aligned}
\{\mathrm{stress}(x)\} &\leftarrow && \text{for } x = 1, 2\\
\mathrm{smokes}(x) &\leftarrow \mathrm{stress}(x) && \text{for } x = 1, 2\\
\{\mathrm{influences}(y, x)\} &\leftarrow && \text{for } x + 1 = y \bmod 2\\
\mathrm{smokes}(x) &\leftarrow \mathrm{influences}(y, x), \mathrm{smokes}(y) && \text{for } x + 1 = y \bmod 2
\end{aligned}$$

This encodes that for person 1 and 2 it is randomly determined whether they are stressed. If they are, they smoke. Furthermore, if one influences the other, which is again random, and smokes, then they also smoke. In order to introduce probabilities, we use algebraic measures.

Definition 1 (Measure). *A (factorized algebraic)* measure $\mu = \langle \Pi, f, F \rangle$ *consists of an answer set program* Π*, a weight function* f *and a set of extensional atoms* $F \subseteq \mathcal{A}(\Pi)$*. The weight* $\mu(\mathcal{I})$ *of an answer set* $\mathcal{I}$ *and the weight* $\mu(a)$ *of* $a \in \mathcal{A}(\Pi)$ *are defined respectively as*

$$\mu(\mathcal{I}) := \prod_{a \in F \cap \mathcal{I}} f(a) \cdot \prod_{a \in F \setminus \mathcal{I}} f(\neg a) \text{ and } \mu(a) := \sum_{\mathcal{I} \in \mathcal{AS}(\Pi), a \in \mathcal{I}} \mu(\mathcal{I}).$$

Measures "measure" values associated with answer set. AASC is then to evaluate $\mu(a)$, i.e., to sum up the weights of all answer sets that satisfy a. We do not focus on any particular application of AASC.

Usually, measures are not necessarily factorized and allow complex terms of sums and products to define the weight of an answer set. Furthermore, they are defined not only for weights over the reals but over any semiring. We stick to the restricted definition above for simplicity.

We can assign probabilities using the measure $\mu_{sm} = \langle \Pi_{sm}, f, F\rangle$, where

$$\begin{aligned} f(\text{stress}(i)) &= 0.4 & f(\neg\text{stress}(i)) &= 0.6 & i &= 1, 2 \\ f(\text{influences}(i,j)) &= 0.3 & f(\neg\text{influences}(i,j)) &= 0.7 & i+1 &\equiv j \bmod 2 \end{aligned}$$

This means that the probability of a person being stressed is 0.4 and the probability that a person influences their friend is 0.3. Therefore, the answer set $\mathcal{I} = \{\text{stress}(1), \text{smokes}(1)\}$ has probability $\mu_{sm}(\mathcal{I}) = 0.4 \cdot 0.6 \cdot 0.7^2$. The query $\mu(\text{smokes}(1))$ corresponds to the probability that person 1 smokes. To evaluate it we need to perform AASC, i.e., sum up the probabilities of all answer sets s.t. $\text{smokes}(1)$ holds.

3. Evaluation Pipeline

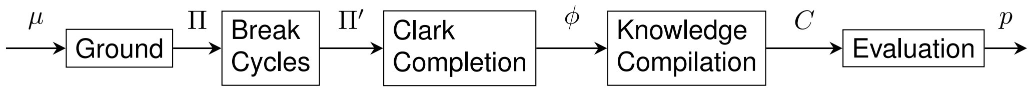


Figure 1: Evaluation pipeline from an algebraic measure μ to the result $\mu(a) = p$ of the query.

Currently, the most promising strategy to perform AASC is the one used by Problog [19]. Here, the general idea is to compile the program Π into an equivalent tractable circuit representation C, like SDD [26] or sd-DNNF [27]. When this succeeds, AASC is possible in linear time in the size of the circuit C. However, while there are so called *knowledge compilers* that perform this task, they can only work on propositional formulas.

The whole pipeline that we and (similarly) Problog use is therefore as in Figure 1. Given an algebraic measure μ, we first ground the logical part of the measure, obtaining a ground program Π. Next, Π is translated into an equivalent program Π', which is *acyclic*, i.e., which does not contain rules with cyclic positive dependencies. For such programs it is known [20] that the Clark completion of Π' results in a propositional formula ϕ, whose models correspond one-to-one to the answer sets of Π'. Thus, by compiling ϕ into a tractable circuit representation C, using a tool like c2d [28], we can obtain the probability p as the final result.

4. Treewidth-Aware Cycle Breaking

Our contribution in [1] focused on the second step in this pipeline, by introducing an algorithm that breaks cycles in a *treewidth-aware* fashion. Finding an equivalent and acyclic program Π' of low treewidth can be beneficial since there are performance guarantees for knowledge compilation based on treewidth [26]. This motivates our main result for cycle-breaking:

Theorem 2. *For any measure $\mu = \langle \Pi, f, F\rangle$, there exists a measure $\mu' = \langle \Pi', f, F\rangle$ with an acyclic program Π' s.t.*

- *for all $a \in \mathcal{A}(\Pi)$ it holds that $\mu(a) = \mu'(a)$,*
- *the treewidth of Π' is bounded by $k \cdot \mathrm{cbs}(\mathrm{DEP}(\Pi))$, where k is the treewidth of Π.*

Here, $\mathrm{cbs}(\mathrm{DEP}(\Pi))$ is a novel parameter, called *component-boosted backdoor size*, which intuitively measures the cyclicity of the dependency graph of Π. It is defined as follows:

Definition 3. *Let G be a digraph. Then, the* component-boosted backdoor size $\mathrm{cbs}(G)$ *is*

- *1, if G is acyclic (which includes $V(G) = \emptyset$)*
- *2, if G is a polytree, i.e. the undirected version of G is connected and acyclic*
- $\max\{\mathrm{cbs}(C) \mid C \in \mathrm{SCC}(G)\}$*, if G is cyclic but not strongly connected*
- $\min\{\mathrm{cbs}(G \setminus S) \cdot (|S| + 1) \mid S \subseteq V(G)\}$ *otherwise*

As the name suggests, *cbs* generalizes the idea of *backdoors*, which have already been considered in the context of ASP [29]. In our context, a backdoor for a digraph G is a vertex set S, such that $G \setminus S$ is a polytree, polyforest or dag. The *backdoor size* of G is the minimum size of a backdoor for G plus 1. Note that our parameter additionally considers that $G \setminus S$ may consist of separate SCCs that can be handled recursively and is therefore bounded by backdoor size.

The proof of Theorem 2 is constructive, in the sense that it allows the formulation of an algorithm that computes Π'. Broadly speaking, the idea is to gradually introduce copies $a^{(1)}, \dots, a^{(\mathrm{cbs}(\mathrm{DEP}(\Pi)))}$ of each atom a that better and better approximate the meaning of a. For this, we copy all rules

$$a \leftarrow b_1, \dots, b_n, \text{not}\, c_1, \dots, \text{not}\, c_m$$

and replace them by

$$a^{(i)} \leftarrow b_1^{(j_1)}, \dots, b_n^{(j_n)}, \text{not}\, c_1, \dots, \text{not}\, c_m,$$

thus expressing the truth of $a^{(i)})$ in terms of already known approximations of $b_1, \dots, b_n$. Theorem 2 guarantees that, if we consider the atoms in the correct order, a fixed point can be reached after considering each atom at most $\mathrm{cbs}(\mathrm{DEP}(\Pi))$ times. We cannot go into details here due to lack of space but refer interested readers to the full paper [1]. Nevertheless, we can apply Theorem 2 to our running example and obtain Π'_{sm} as

$$\begin{aligned}
\{\mathrm{stress}(x)\} &\leftarrow \text{ for } x = 1, 2\\
\{\mathrm{influences}(y, x)\} &\leftarrow \text{ for } x + 1 = y \bmod 2
\end{aligned}$$

$$\begin{array}{llll}
\mathrm{smokes}(1)^1 & \leftarrow \mathrm{stress}(1) & \mathrm{smokes}(1)^1 & \leftarrow \mathrm{influences}(2,1), \bot\\
\mathrm{smokes}(2) & \leftarrow \mathrm{stress}(2) & \mathrm{smokes}(2) & \leftarrow \mathrm{influences}(1,2), \mathrm{smokes}(1)^1\\
\mathrm{smokes}(1) & \leftarrow \mathrm{stress}(1) & \mathrm{smokes}(1) & \leftarrow \mathrm{influences}(2,1), \mathrm{smokes}(2)
\end{array}$$

Observe, that Π'_{sm} has the same answer sets with respect to the original atoms and is acyclic. Also the treewidth of Π'_{sm} has increased by at most 1, since only the atom $\mathrm{smokes}(1)^1$ was added.

Comparison to other Cycle-Breaking Algorithms

There exists a wide variety of algorithms for cycle-breaking in the ASP literature [24, 23, 22, 30]. However, since most ideas were originally intended for ASP, where it suffices to find one answer set, some of these algorithms do not preserve the original models bijectively [22, 30].

The remaining two ideas preserve answer sets bijectively. As such, Janhunen's [24] was considered for the implementation of Problog (cf. [31]) and Mantadeli's and Janssen's [23] is even still part of the standard Problog implementation. While the original papers did not consider the effects that these translations have on treewidth, the strategy to bound treewidth used in [1, 22] can be applied to these translation as well. This results in treewidth upper-bounds of $k \log_2(C_{\max})$ and kZ, respectively, where $C_{\max}$ is the size of the largest strongly connected component of the dependency graph of Π and Z is the largest number of simple (directed) cycles that any atom is contained in.

One can show that $cbs(.)$ is always bounded by Z and possible exponentially smaller, since on a clique of size n, we have $Z \geq 2^{n-1}$, whereas $cbs(.)$ of a clique of size n is n. Therefore, Theorem 2 provides strictly better treewidth bounds in general. Observe, however, that for the first translation, the factor grows *logarithmically* in $C_{\max}$. It follows that for a clique of size n the upper-bound $k \log_2(n)$ of the first translation is exponentially lower than kn, which is our bound. On the other hand for a polytree T with n vertices $cbs(T) = 2$, whereas $\log_2(n)$ is not constant.

5. Implementation & Experiments

We implemented a system that adheres to the evaluation procedure above, resulting in the prototypical solver aspmc[1]. aspmc is written in Python3 and allows for Problog programs Π as input in order to compute the answers to probabilistic (resp. algebraic) queries in Π.

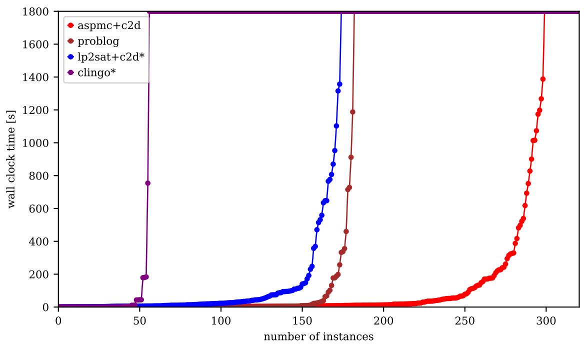
Benchmark Setting. In order to evaluate the performance of aspmc, we compare the computation of probabilities for atoms of Problog programs. For solvers that are not able to evaluate probabilistic queries we compute the number of answer sets.

Compared Solvers. In our experiments, we mainly compare against Problog 2.1.0.42 with the arguments "-k sdd", clingo 5.4.0, where we used arguments "-q -n 0" in order to count answer sets, as well as lp2sat+c2d, where instances are translated [32] to CNFs by lp2normal 2.18 in combination with lp2atomic 1.17 and lp2sat 1.24 with answer sets being counted with c2d version 2.2 [28]. Our system is aspmc+c2d, which breaks the cycles and performs AASC on a tractable circuit representation of the constructed CNF, obtained via c2d 2.2.

Benchmark Instances. The instances we used are the benchmarks from [33] and were kindly provided to us by the authors via personal communication. They adhere to typical benchmark domains consisting of 490 instances of the standard smoker's example, 50 instances of the gene's problem [34] and 63 web knowledge base instances [35].

Benchmark Platform. All our solvers ran on a cluster consisting of 12 nodes. Each node of the cluster is equipped with two Intel Xeon E5-2650 CPUs, where each of these 12 physical

[1]available at github.com/raki123/aspmc (open source).



solver configuration	$\sum$	tw ranges 0-3	4-6	>6	unique	time[h]
aspmc+c2d	**299**	**184**	**113**	2	**116**	**133.44**
Problog	182	168	12	2	0	187.41
lp2sat+c2d*	174	167	3	**4**	2	194.42
clingo*	56	56	0	0	0	248.92

Figure 2: Cactus plot of the different solver configurations (left); x-axis shows the number of solved instances and the y-axis depicts runtimes sorted for each configuration individually in ascending order. Detailed results (right): “Σ” is the number of solved instances in total; “tw ranges” shows the number of solved instances grouped by treewidth upper bounds; “unique” refers to the number of instances solved only by that configuration. Finally, “time[h]” is the total runtime over all instances in hours, including timeouts. Configurations marked with “*” refer to counting solutions.

cores runs at 2.2 GHz clock speed and has access to 256 GB shared RAM. Results are gathered on Ubuntu 16.04.1 powered on kernel 4.4.0-139 with hyperthreading disabled and Python 3.7.6.

Experimental Results & Discussion. In our evaluation, we mainly compare total wall clock time and number of solved instances. Concerning benchmark limits, we consider a timeout of 1800 seconds and an 8 GB RAM limit per instance and solver.

Figure 2 (left) shows a plot over all instances, which indicates that aspmc+c2d solves more instances faster than any of the other configurations that we benchmarked. The detailed results in Figure 2 (right) indicate that cycle breaking works well for probabilistic reasoning.

Other experiments [1] showed that clingo is extremely fast at enumerating solutions. Thus, on instances with not that many solutions, clingo is faster than any of the compilation-based approaches we considered. On the other hand, when there are more solutions, considering each of them once as in enumeration, is outperformed by the compilation-based approaches lp2sat+c2d, aspmc+c2d and Problog that solve significantly more instances in this case.

6. Conclusion and Future Work

For any program Π, there is an equivalent acyclic program, whose treewidth is at most $\mathrm{cbs}(\mathrm{DEP}(\Pi))$ times bigger, where $\mathrm{cbs}(.)$ is a novel parameter that measures the cyclicity of directed graphs. The bound on the treewidth provides us with worst case guarantees for the knowledge compilation step in AASC. Our experimental evaluation of the prototype implementation aspmc shows that this idea does not only provide interesting theoretical results but provides a significant speedup on standard Problog benchmarks compared to other solvers.

Acknowledgements

The work is supported by Austrian Science Fund (FWF), Grants Y698, P32830 and W1255-N23, the Vienna Science and Technology Fund, Grant WWTF ICT19-065.

References

[1] T. Eiter, M. Hecher, R. Kiesel, Treewidth-aware cycle-breaking for algebraic answer set counting, in: 18th International Conference on Principles of Knowledge Representation and Reasoning, 2021. To Appear. Preliminary version available at https://drive.google.com/file/d/1XvVCxNqssJKj18TghztkOUVDAsne5R_m/view?usp=sharing.

[2] J. Lee, Z. Yang, LPMLN, weak constraints, and P-log, in: Thirty-First AAAI Conference on Artificial Intelligence, 2017.

[3] C. Baral, M. Gelfond, N. Rushton, Probabilistic reasoning with answer sets, Theory and Practice of Logic Programming 9 (2009) 57–144.

[4] M. Nickles, A. Mileo, Web stream reasoning using probabilistic answer set programming, in: International Conference on Web Reasoning and Rule Systems, Springer, 2014, pp. 197–205.

[5] F. G. Cozman, D. D. Mauá, The joy of probabilistic answer set programming: Semantics, complexity, expressivity, inference, International Journal of Approximate Reasoning 125 (2020) 218–239.

[6] L. De Raedt, A. Kimmig, H. Toivonen, Problog: A probabilistic prolog and its application in link discovery., in: IJCAI, volume 7, Hyderabad, 2007, pp. 2462–2467.

[7] G. Brewka, J. Delgrande, J. Romero, T. Schaub, asprin: Customizing answer set preferences without a headache, in: Twenty-Ninth AAAI Conference on Artificial Intelligence, 2015.

[8] A. Kimmig, G. Van den Broeck, L. De Raedt, An algebraic prolog for reasoning about possible worlds, in: Twenty-Fifth AAAI Conference on Artificial Intelligence, 2011.

[9] T. Eiter, R. Kiesel, Weighted LARS for quantitative stream reasoning, in: Proc. ECAI'20, 2020.

[10] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Clingo= ASP+ control: Preliminary report, arXiv preprint arXiv:1405.3694 (2014).

[11] J. K. Fichte, M. Hecher, M. Morak, S. Woltran, Dynasp2. 5: Dynamic programming on tree decompositions in action, in: International Symposium on Parameterized and Exact Computation (IPEC), volume 89 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, pp. 17:1–17:13.

[12] D. R. G. KU Leuven, Problog, https://github.com/ML-KULeuven/problog, 2021.

[13] A. Kimmig, G. Van den Broeck, L. De Raedt, Algebraic model counting, Journal of Applied Logic 22 (2017) 46–62.

[14] U. Oztok, A. Darwiche, A top-down compiler for sentential decision diagrams, in: Twenty-Fourth International Joint Conference on Artificial Intelligence, 2015.

[15] T. Sang, P. Beame, H. A. Kautz, Performing bayesian inference by weighted model counting, in: AAAI, volume 5, 2005, pp. 475–481.

[16] M. Thurley, sharpSAT–counting models with advanced component caching and implicit

BCP, in: International Conference on Theory and Applications of Satisfiability Testing, Springer, 2006, pp. 424–429.

[17] D. Tuckey, A. Russo, K. Broda, Pasocs: A parallel approximate solver for probabilistic logic programs under the credal semantics, arXiv preprint arXiv:2105.10908 (2021).

[18] M. Nickles, diff-sat–a software for sampling and probabilistic reasoning for sat and answer set programming, arXiv preprint arXiv:2101.00589 (2021).

[19] D. Fierens, G. Van den Broeck, J. Renkens, D. Shterionov, B. Gutmann, I. Thon, G. Janssens, L. De Raedt, Inference and learning in probabilistic logic programs using weighted boolean formulas, Theory and Practice of Logic Programming 15 (2015) 358–401.

[20] F. Fages, Consistency of Clark's completion and existence of stable models, Journal of Methods of logic in computer science 1 (1994) 51–60.

[21] J. Vlasselaer, G. Van den Broeck, A. Kimmig, W. Meert, L. De Raedt, Tp-compilation for inference in probabilistic logic programs, International Journal of Approximate Reasoning 78 (2016) 15–32.

[22] M. Hecher, Treewidth-aware Reductions of Normal ASP to SAT - Is Normal ASP Harder than SAT after All?, in: Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, 2020, pp. 485–495. URL: https://doi.org/10.24963/kr.2020/49. doi:10.24963/kr.2020/49.

[23] T. Mantadelis, G. Janssens, Dedicated tabling for a probabilistic setting, in: Technical Communications of the 26th International Conference on Logic Programming, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[24] T. Janhunen, Representing normal programs with clauses, in: ECAI, volume 16, Citeseer, 2004, p. 358.

[25] T. Eiter, G. Ianni, T. Krennwallner, Answer set programming: A primer, in: Reasoning Web International Summer School, Springer, 2009, pp. 40–110.

[26] A. Darwiche, Sdd: A new canonical representation of propositional knowledge bases, in: Twenty-Second International Joint Conference on Artificial Intelligence, 2011.

[27] A. Darwiche, P. Marquis, A knowledge compilation map, Journal of Artificial Intelligence Research 17 (2002) 229–264.

[28] A. Darwiche, New advances in compiling CNF into decomposable negation normal form, in: ECAI, IOS Press, 2004, pp. 328–332.

[29] J. K. Fichte, S. Szeider, Backdoors to tractable answer set programming, Artif. Intell. 220 (2015) 64–103. URL: https://doi.org/10.1016/j.artint.2014.12.001. doi:10.1016/j.artint.2014.12.001.

[30] F. Lin, J. Zhao, On tight logic programs and yet another translation from normal logic programs to propositional logic, in: International Joint Conference on Artificial Intelligence, 2003.

[31] D. Fierens, G. Van den Broeck, I. Thon, B. Gutmann, L. D. Raedt, Inference in probabilistic logic programs using weighted cnf's, in: Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence, 2011, pp. 211–220.

[32] J. Bomanson, lp2normal - A normalization tool for extended logic programs, in: LPNMR, volume 10377 of *Lecture Notes in Computer Science*, Springer, 2017, pp. 222–228.

[33] E. Tsamoura, V. Gutiérrez-Basulto, A. Kimmig, Beyond the Grounding Bottleneck: Datalog Techniques for Inference in Probabilistic Logic Programs, in: Conference on Artificial

Intelligence (AAAI), AAAI Press, 2020, pp. 10284–10291. URL: https://aaai.org/ojs/index.php/AAAI/article/view/6591.

[34] O. Ourfali, T. Shlomi, T. Ideker, E. Ruppin, R. Sharan, Spine: a framework for signaling-regulatory pathway inference from cause-effect experiments, Bioinformatics 23 (2007) i359–i366.

[35] J. Davis, P. Domingos, Deep transfer via second-order markov logic, in: Proceedings of the 26th annual international conference on machine learning, 2009, pp. 217–224.