LP2NORMAL — A Normalization Tool for Extended Logic Programs

Jori Bomanson(✉)

Department of Computer Science, Aalto University, Espoo, Finland
jori.bomanson@aalto.fi

Abstract. Answer set programming (ASP) features a rich rule-based modeling language for encoding search problems. While normal rules form the simplest rule type in the language, various forms of extended rules have been introduced in order to ease modeling of complex conditions and constraints. Normalization means replacing such extended rules with identically functioning sets of normal rules. In this system description, we present LP2NORMAL, which is a state-of-the-art normalizer that acts as a filter on ground logic programs produced by grounders, such as GRINGO. It provides options to translate away choice rules, cardinality rules, and weight rules, and to rewrite optimization statements using comparable techniques. The produced logic programs are suitable inputs to tools that lack support for extended rules, in particular. We give an overview of the normalization techniques currently supported by the tool and summarize its features. Moreover, we discuss the typical application scenarios of normalization, such as when implementing the search for answer sets using a back-end solver without direct support for cardinality constraints or pseudo-Boolean constraints.

1 Introduction

Answer set programming (ASP) [9,15] is a declarative programming paradigm that features a rich rule-based modeling language for encoding search problems. Normal rules form the base fragment of the language and their declarative interpretation is based on the notion of *answer sets* also known as *stable models* [12]. In order to ease modeling of complex conditions and constraints, different syntactic extensions have been introduced. In particular, the *extended rule types* of [16], i.e., *choice*, *cardinality*, and *weight* rules and *optimization* statements are central primitives for modeling in ASP [11].

When it comes to implementing language extensions, there are two basic strategies. The first is to extend the underlying answer-set solver to natively handle extended syntax. The second is to treat extensions as syntactic sugar and translate them away. The tool described in this paper is motivated by the latter *translation-based* strategy in a setting where the used ASP toolchain does not support extended rules. The *normalization* [14] of such rules means replacing them with sets of normal rules that are semantically indistinguishable from them in any context. For example, the weight rule a :- $5 \leq \#\text{sum}\,\{2 : b; 3 : c; 5 : \text{not } d\}$

M. Balduccini and T. Janhunen (Eds.): LPNMR 2017, LNAI 10377, pp. 222–228, 2017.
DOI: 10.1007/978-3-319-61660-5_20

can be normalized into a :- b, c and a :- not d. While the normalization of choice rules is straightforward, sophisticated normalization techniques for other extended rules have been developed in a trilogy of papers [6–8].

In this system description, we present a tool called LP2NORMAL (2.27)[1], which is a state-of-the-art normalizer that can be used to filter ground logic programs produced by a grounder before forwarding the programs to a solver for the computation of answer sets. It ships with the functionality required to translate away extended rules, by which we exclusively refer to choice, cardinality, and weight rules in the sequel, as well as to *rewrite* optimization statements in an analogous way.

The rest of this paper is organized as follows. In Sect. 2, we give a brief overview of translation techniques that can be exploited in the process of normalizing extended rule types. Specific features of LP2NORMAL available through numerous command-line options are summarized in Sect. 3. Some existing application scenarios of normalization are discussed in Sect. 4. Finally, we conclude the paper in Sect. 5.

2 Overview of Normalization Techniques

In this section, we look into techniques for normalizing cardinality and weight rules. We also describe methods for simplifying the rules before normalization and the idea of a *cone of influence* for pruning the normalized output. Finally, we turn to the task of rewriting optimization statements. The underlying translation schemes to be described can be interpreted as specifications of counting and comparison operations between numbers expressed in unary, binary, or even mixed-radix bases.

In ASP, to say that at least k out of n literals are satisfied one typically uses a single *cardinality rule*. This condition is also expressible in a number of normal rules, such as in $O(n(n-k))$ rules and auxiliary atoms that form a Binary Decision Diagram (BDD) [10] or a *counting grid* [14,16]. The rows of this grid are built in a dynamic programming fashion based on the previous rows, and each row encodes a partial count of satisfied input literals as a unary number. Translation schemes identical or close to this involve the Sinz counter [17] and sequential counters (SEQ) [13]. More concise encodings are obtainable via merge sorting, where again partial counts are built up of smaller counts, but this time by recursively merging halves of the input. We obtain an ASP encoding of an odd-even sorting network of size $O(n(\log n)^2)$ by using odd-even merging networks [5], and expressing their building blocks, i.e., comparators, in three normal rules each. On the other hand, we obtain a *totalizer* of size $O(n^2)$ by using direct mergers that require no additional auxiliary variables [4].

The translations mentioned so far can be encoded without introducing negated literals beyond those in the input. With the use of additional negation, even more concise schemes are attainable. For example, an encoding of a

[1] Available at http://research.ics.aalto.fi/software/asp.

binary adder yields no more than $O(n \log k)$ rules and atoms. However, analogous encodings have proven difficult in SAT solving due to their poor propagation properties. Moreover, they may lead to soundness issues in ASP, where negation and positive recursion require extra care.

Lower bounds in cardinality rules, such as k above, can be replaced or complemented by upper bounds in typical ASP systems, which convert the latter to the former in the grounding phase [16]. Hence, only lower bounds remain for normalization. Similarly, syntactic constructions that combine choice and cardinality rules can be interpreted as short hands for the two types of rules, which can be normalized separately.

When it comes to *weight rules*, in which literals are generalized to have weights w_i, we may apply simplifications prior to normalization, potentially reducing numbers of literals or their weights. Both of these outcomes generally lower the size of the subsequent normalization. Among these simplifications, we have some basic ones such as removal of literals with large weights $w_i \geq k$ that can be compensated with simple normal rules. Also, we may factor out any common divisor d of the weights and divide the bound, rounding it up. Furthermore, this division is applicable even when one of the weights w_i results in a remainder, as long as the corresponding quotient is afterward incremented by one in the case that $k \leq w_i \pmod d$. Other scenarios providing simplification opportunities include cases where a number of the largest weights in a rule are always required in order to reach the bound; where a pair of weights together satisfy the bound; where a weight is too small to ever make a difference; and certain cases where analysis of the residues resulting from division of the bound and weights with a heuristically chosen divisor reveals that the division can be done given minor adjustments to some of the numbers.

The actual normalization techniques in the tool for weight rules mainly revolve around two types of translations. On the one hand, we have sequential weight counters (SWC) [13] and Reduced-Ordered Binary Decision Diagrams (ROBDDs) [1], both of the size $O(nk)$. They are particularly compact for small rules, for which the asymptotic size is not relevant in practice. On the other hand, we have sorting and merging based normalizations, which are $O(n(\log n)^2 \log w_{\max})$ in size, e.g., when odd-even mergers are used, where $w_{\max}$ denotes the largest input weight [6,10]. In the constructions, there are $c \leq \log_2 w_{\max}$ sorters, each of which intuitively counts digits of a certain significance, which are followed by $c-1$ mergers that perform deferred carry propagation. These normalizations can be compressed with *structure sharing* [6]. This sharing method stems from the observation that the sets of inputs to the sorters overlap significantly in general. When merge sorters are used, the overlap leads to duplication in their structure, which may be maximized via optimization and then eliminated.

In normalizations of cardinality and weight rules, there is only a single output atom per rule that we are interested in. Yet, in their basic form, the outlined normalization strategies, with the exception of those based on (RO)BDDs, define sequences of outputs that are not all needed. Namely, counting grids, mergers,

sorters, and SWCs produce entire vectors of atoms with sorted truth values. Now one may imagine that we mark a single output, and propagate this information of what is wanted and what is not backward through the rules in the translation, so as to compute what we call a *cone of influence*. Then, the actual normalization may be produced in a forward phase, where only the rules defining atoms that fell inside the cone are included. From our practical experience, this pruning technique is important for sorters, and sorter based translations as well as SWCs. Moreover, it also brings down the asymptotic size of odd-even merge sorting programs, which it prunes down from a size of $O(n(\log n)^2)$ to *selection programs* of size $O(n(\log m)^2)$, where m is the lesser of the bound k and $n - k + 1$. The resulting size rivals that of selection network designs used in SAT [2].

Whereas the above options concern normalization resulting in purely normal rules, LP2NORMAL also supports *rewriting of optimization statements* using techniques similar to those used in normalization, but which generate modified optimization statements in adddition to normal rules. The rewritings generally define auxiliary atoms using normal rules so as to encode the sum, or some partial sums of the weighted literals that make up an optimization statement. Such a statement is then replaced by one or more statements specified in terms of the new auxiliary atoms. These rewritings are solely aimed at boosting solving performance by offering new atoms to branch on and to use in learnt nogoods. However, these rewritings may grow impractically large in terms of the generated atoms and rules, especially when applied to optimization statements that carry substantial amounts of information. To alleviate this issue, we have developed ways to limit the size increase via refined control over how much rewriting is done.

3 Implementation

In this section, we cover the usage and highlight some implementation details of the normalizer tool LP2NORMAL. A summary of the discussed options is shown in Table 1.

By default, all extended rules are translated into normal rules, while leaving other rules and statements intact. This behaviour is configurable via command line options, which are prefixed with `-c`, `-w`, and `-o` for cardinality, weight, and optimization statements, respectively. Rules can be kept as they are with the option `-k` for choice rules and `-ck`, `-wk`, `-ok` for the rest.

Cardinality rules are by default normalized using an automatic scheme `-cc` that generates merge sorting programs built recursively of direct mergers and odd-even mergers. The choice between them is based on trial runs that reveal which one introduces fewer atoms and rules. The decision can be fixed to direct mergers with `-ct` and to odd-even mergers with `-ch`. Moreover, an option `-cs` is available for basing the normalization on selection networks instead.

For weight rules, one may pick the schemes based on SWCs with `-wc` and ROBDDs with `-wb`. Weight rule translations constructed from sorters and mergers are primarily controlled with the option `-wq`. Due to a design choice in

Table 1. Command-line options of translations in LP2NORMAL by extended rule type and with asymptotic sizes after cone-of-influence simplification. Here $m = \min\{k, n-k+1\}$.

	Options	Rules (Atoms)	
Choice		$O(n)$	
Cardinality	`-cn`	$O(n(n-k))$	Counting grid [10,14,16]
Cardinality	`-ct`	$O(nm)$ $(O(n \log m))$	Totalizer [4]
Cardinality	`-ch`	$O(n(\log m)^2)$	Odd-even merge sort [5,8,10]
Cardinality	`-cc`	$O(n(\log m)^2)$	Automatic merge sort
Weight	`-wc`	$O(nk)$	SWC [13]
Weight	`-wb`	$O(nk)$	ROBDD [1]
Weight	`-wq -cc`	$O(n(\log n)^2 \log w_{\max})$	Network of merge sorters [3,6,10]
Optimization	`-oqn -cc`	$O(n(\log n)^2 \log w_{\max})$	Network of merge sorters [7]
Optimization	`-oKp`q `-oqn -cc`	$O(nq(\log n)^2)$	Rewrite q most significant digits [7]

LP2NORMAL aiming for simplicity, the choice of sorters and mergers used here are inferred from any active cardinality rule options. That is, if a user requests cardinality rules to be translated using odd-even mergers and sorters with `-ch` and weight rules with `-wq`, then those types of mergers and sorters are used in the weight rule normalizations as well.

For optimization statements, the option `-oqn` instructs LP2NORMAL to proceed with optimization statements in the same way as when it applies `-wq` to weight rules, but with the following difference. The translation is cut short before a bound check is encoded, and instead the atoms that the bound check would have depended on are printed in an optimization statement. Moreover, there is an option `-ox` to use certain weight rule translation techniques, primarily those based on SWCs or ROBDDs, to produce a single sorted and weighted sequence of atoms that encodes all subset sums of the contents of an optimization statement. This option is not always feasible, but it serves as a proof-of-concept for how to use these weight rule techniques to rewrite optimization statements in a natural way. Finally, we highlight a collection of options prefixed by `-oK` or `-ok` that select parts of optimization statements to be rewritten or kept from being rewritten. For example, with `-oKp3` one may instruct LP2NORMAL to first split every weight after three of its most significant digits and then to apply any specified rewriting, such as `-oqn`, to the more significant part only.

4 Applications

Our main driving motivation for normalization in ASP has been to add support for extended rules to solvers that would otherwise accept only normal rules.

To this end, LP2NORMAL took part in several systems submitted by the Aalto team to the Sixth Answer Set Programming Competition[2]. The following is an example of a pipeline relying on the SAT solver LINGELING for solving ASP problems. In the pipeline, LP2NORMAL translates away extended rules, while the tools in the middle take care of translating the resulting normal rules to SAT.

```
lp2normal | lp2acyc | lp2sat -b | lingeling
```

In the above, we could alternatively use normalization capabilities of the state-of-the-art ASP solver CLASP (3.2.2) via its options `--pre` and `--trans-ext=all`. However, the techniques implemented therein generally yield larger output. Figure 1 depicts the case for cardinality rules on $n \in \{25, 50, 100\}$ atoms. As another use case, one of the systems in the competition combines LP2NORMAL and CLASP in a configuration where the normalizer translates cardinality and weight rules of only modest size, and rewrites parts of optimization statements. In this case, the role of the normalizer is to alter the solving performance of the solver, which can handle extended rules natively as well. The impact on performance varies benchmark by benchmark, which frequently improves but also sometimes degrades.

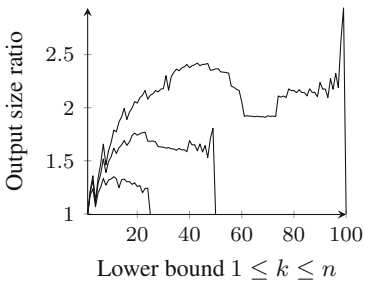

Fig. 1. Ratios of the numbers of integers in the normalizations produced by CLASP in comparison to LP2NORMAL.

5 Conclusion

We summarized the most important capabilities of the tool LP2NORMAL concerning the normalization of cardinality and weight rules and rewriting of optimization statements in answer set programs. In future development, we plan to incorporate optimal sorting networks into the tool together with other compact networks generated offline for small, fixed ranges of input parameters. Finally, we continue to explore partial rewriting, which has proven to be an effective way to limit translation size and enhance optimization performance [7].

References

1. Abío, I., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Mayer-Eichberger, V.: A new look at BDDs for pseudo-Boolean constraints. J. Artif. Intell. Res. **45**, 443–480 (2012)

[2] Participant systems are available at http://aspcomp2015.dibris.unige.it/participants.

2. Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Cardinality networks: a theoretical and empirical study. Constraints **16**(2), 195–221 (2011)
3. Bailleux, O., Boufkhad, Y., Roussel, O.: New encodings of pseudo-Boolean constraints into CNF. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 181–194. Springer, Heidelberg (2009). doi:10.1007/978-3-642-02777-2_19
4. Bailleux, O., Boufkhad, Y.: Efficient CNF encoding of Boolean cardinality constraints. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 108–122. Springer, Heidelberg (2003). doi:10.1007/978-3-540-45193-8_8
5. Batcher, K.: Sorting networks and their applications. In: AFIPS Spring Joint Computer Conference, pp. 307–314. ACM (1968)
6. Bomanson, J., Gebser, M., Janhunen, T.: Improving the normalization of weight rules in answer set programs. In: Fermé, E., Leite, J. (eds.) JELIA 2014. LNCS (LNAI), vol. 8761, pp. 166–180. Springer, Cham (2014). doi:10.1007/978-3-319-11558-0_12
7. Bomanson, J., Gebser, M., Janhunen, T.: Rewriting optimization statements in answer-set programs. In: Technical Communications of ICLP 2016, vol. 52, OASIcs, pp. 5:1–5:15 (2016)
8. Bomanson, J., Janhunen, T.: Normalizing cardinality rules using merging and sorting constructions. In: Cabalar, P., Son, T.C. (eds.) LPNMR 2013. LNCS (LNAI), vol. 8148, pp. 187–199. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40564-8_19
9. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. Commun. ACM **54**(12), 92–103 (2011)
10. Eén, N., Sörensson, N.: Translating pseudo-Boolean constraints into SAT. J. Satisfiability Boolean Model. Comput. **2**, 1–26 (2006)
11. Gebser, M., Schaub, T.: Modeling and language extensions. AI Mag. **37**(3), 33–44 (2016)
12. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of ICLP 1988, pp. 1070–1080. MIT Press (1988)
13. Hölldobler, S., Manthey, N., Steinke, P.: A compact encoding of pseudo-Boolean constraints into SAT. In: Glimm, B., Krüger, A. (eds.) KI 2012. LNCS (LNAI), vol. 7526, pp. 107–118. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33347-7_10
14. Janhunen, T., Niemelä, I.: Compact translations of non-disjunctive answer set programs to propositional clauses. In: Balduccini, M., Son, T.C. (eds.) Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning. LNCS (LNAI), vol. 6565, pp. 111–130. Springer, Heidelberg (2011). doi:10.1007/978-3-642-20832-4_8
15. Janhunen, T., Niemelä, I.: The answer set programming paradigm. AI Mag. **37**(3), 13–24 (2016)
16. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artif. Intell. **138**(1–2), 181–234 (2002)
17. Sinz, C.: Towards an optimal CNF encoding of Boolean cardinality constraints. In: Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 827–831. Springer, Heidelberg (2005). doi:10.1007/11564751_73