An Experiment on Tabled Evaluation for Hidden Predicates[⋆]

Pedro Cabalar and Martín Diéguez

Department of Computer Science,
University of Corunna (Spain)
{cabalar,martin.dieguez}@udc.es

Abstract. Most Answer Set Programming grounders allow filtering the information shown in each answer set by selecting which predicates must be shown or become hidden instead. Full grounding of hidden predicates (usually introduced for auxiliary computations) is in many cases not necessary. In the paper, we consider the possibility of avoiding grounding of hidden (stratified) predicates by using tabled evaluation: a top-down strategy combined with an extension table that records the previously solved goals.

1 Introduction

Answer Set Programming (ASP) [1, 2] has recently become one of the most popular and successful paradigms for Nonmonotonic Reasoning (NMR) and Knowledge Representation (KR). Part of this success is due to the simplicity and robustness of its theoretical foundations which rely on the notion of *answer set* or *stable model* [3]. Originally defined for normal logic programs, this semantics proved to be flexible enough to end up covering the general syntax of arbitrary first order theories [4, 5] and accommodating useful operators for practical KR like weight constraints [6] or aggregates [7, 8]. A second, but not less important factor for the success of ASP has been the availability of efficient solvers that were boosted both in number and performance, thanks to the establishment of a competition [9] with public benchmarks and results. Most ASP solvers divide their computation into two differentiate steps: in a first phase, called *grounding*, variables in each rule are replaced by all their possible combinations of ground instances (the constants in the program). In a second step, the solver computes answer sets for the resulting ground program. Some tools like `lparse`[1] or `gringo`[2] are distributed as separate grounders that must be combined with solvers that exclusively accept propositional programs as an input, like `smodels`[3], `clasp`[4],

⋆ This research was partially supported by Spanish MEC project TIN2009-14562-C05-04 and Xunta de Galicia project INCITE08-PXIB105159PR.

[1] `http://www.tcs.hut.fi/Software/smodels/`
[2] `http://sourceforge.net/projects/potassco/files/gringo/`
[3] `http://www.tcs.hut.fi/Software/smodels/`
[4] `http://sourceforge.net/projects/potassco/files/clasp/`

`cmodels`[5] or `assat`[6]. Other tools like `DLV`[7] embody the grounder and the propositional solver in a same package.

Although the input languages of `lparse`, `gringo` and `DLV` have some differences, the three tools allow selecting which predicates are shown in the obtained stable models and which ones can be hidden. Typically, these hidden predicates contain intermediate or auxiliary information that is actually irrelevant for describing the final solutions of the problem we are interested in. The three mentioned ASP grounders use this feature as a simple output choice, so that when a predicate is hidden, it is just filtered out from the information displayed in an answer set, but its grounding and computation is not actually affected. However, the following question arises: once we know that the information provided by a predicate will be irrelevant, is it always necessary to compute its full extent?

In this paper we consider a partial grounding of hidden predicates, so that their extent is computed depending on the values of variables fixed by other predicates. In this way, the hidden predicate is called as a top-down query rather than computing its full extent using a bottom-up strategy. In order to avoid repeating subgoals due to multiple recursive calls, instead of a pure top-down strategy, we propose applying a technique called *tabled evaluation* [10] that combines top-down queries with an extension table that keeps record of previously solved goals. In fact, this technique has been implemented and extensively used in the `XSB`[8] Prolog interpreter, which has been used as a back-end in our preliminary experiments.

The rest of the paper is organised as follows. In the next section we present a motivating example that is followed in Section 3 by a description of the prototype we have implemented and a preliminary experiment, commenting the obtained results. Section 4 contains a brief discussion and concludes the paper.

2 A Motivating Example

To illustrate our main purpose, consider the following motivating example.

Example 1. Suppose we have a directed graph where some nodes are marked as *source* and that we want to generate arbitrary sets of nodes that are reachable from a source. To this aim, we include a predicate $in(X)$ that points out that node X is in our current selection.

[5] `http://www.cs.utexas.edu/users/tag/cmodels.html`
[6] `http://assat.cs.ust.hk/`
[7] `http:www.dbai.tuwien.ac.at/proj/dlv/`
[8] `http://xsb.sourceforge.net/`

A possible program that solves this problem and is syntactically accepted by the three grounders mentioned before is the following one:

$$\begin{aligned} reach(X,Y) &\leftarrow edge(X,Y) && (1)\\ reach(X,Z) &\leftarrow node(X), reach(X,Y), edge(Y,Z) && (2)\\ in(Y) &\leftarrow source(X), reach(X,Y), not\ out(Y) && (3)\\ out(Y) &\leftarrow source(X), reach(X,Y), not\ in(Y) && (4) \end{aligned}$$

As usual in ASP, the cyclic rules (3)-(4) involving an auxiliary predicate $out(X)$, complement of $in(X)$, are used to generate possible solutions. Each instance of this problem would be given as a set of facts for predicates $node(X)$, $edge(X,Z)$ and $source(X)$. For instance, the graph depicted in Figure 1 (double circled nodes are sources) would be represented by the set of facts:

$$\begin{array}{r} node(1..9).\ source(3).\ source(5).\\ edge(1,2).\ edge(2,3).\ edge(1,4).\ edge(2,5).\\ edge(3,6).\ edge(4,5).\ edge(5,6).\ edge(4,7).\\ edge(5,8).\ edge(6,9).\ edge(7,8).\ edge(8,9). \end{array}$$

As only three nodes, 6, 8 and 9, are reachable from source nodes, it is easy to check that the resulting program has eight answer sets, that correspond to all solutions, i.e., all possible subsets of $\{in(6), in(8), in(9)\}$. The complete answer sets, however, will contain much more information than the extent for predicate $in(X)$. In particular, each answer set will contain the facts we entered to describe the graph plus the extent of predicates $reach(X,Y)$ and $out(X)$. For instance, all answer sets will include the following 27 facts for predicate $reach(X,Y)$:

$$\begin{array}{r} reach(1,2), reach(1,3), reach(1,4), reach(1,5), reach(1,6), reach(1,7),\\ reach(1,8), reach(1,9), reach(2,3), reach(2,5), reach(2,6), reach(2,8),\\ reach(2,9), reach(3,6), reach(3,9), reach(4,5), reach(4,6), reach(4,7),\\ reach(4,8), reach(4,9), reach(5,6), reach(5,8), reach(5,9), reach(6,9),\\ reach(7,8), reach(7,9), reach(8,9) \end{array}$$

Since our interest is focused on the solutions in terms of $in(X)$, we would typically hide the rest of predicates. In `lparse` and `gringo` this is done by including the clauses:

$$\begin{array}{l} hide.\\ show\ in(X). \end{array}$$

whereas in `DLV` we would include the command line option `-filter=in` for the same purpose. In the three grounders, hiding the extent of a predicate does not have any real implication on the grounding or computation of the answer sets: it just filters the facts that are shown in the output, but all the information is computed anyway. However, if we look carefully at our example, it can be noticed that computing reachable nodes from 1, 2, 4 and 7 could be avoided if we

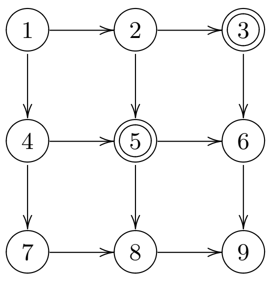

Fig. 1. An example of directed graph with two "*source*" nodes.

just started from source nodes 3 and 5 and followed predicate *reach* on demand. For instance, once we fix $X = 3$ in rule (3) as one of the possible values obtained from $source(X)$, we would make a call to $reach(3, Y)$ and try to solve it using a top-down strategy. By recursive applications of (1)-(2) we would then obtain the values $Y = 6$ and $Y = 9$, which would finally allow us generating the ground rules:

$$in(6) \leftarrow not\ out(6)$$
$$in(9) \leftarrow not\ out(9)$$

Using this strategy, we would never need computing facts for $reach(1, Y)$, $reach(2, Y)$, $reach(4, Y)$ or $reach(7, Y)$, something that reduces the 27 facts obtained before to only the following 7:

$$reach(3, 6), reach(3, 9), reach(5, 6), reach(5, 8),$$
$$reach(5, 9), reach(6, 9), reach(8, 9)$$

A pure top-down evaluation, however, introduces a new problem, not present in bottom-up computation: some goals are called more than once and its computation must be repeated. As a simple example, suppose we marked node 1 as source, making $source(1)$ true. Then, we would be continuously repeating goals: for instance, $reach(6, 9)$ would also be called in the computation of each $reach(Y, 9)$ for all nodes $Y = 1, \ldots, 5$, that is, those that pass through node 6 to reach 9. In an example like that, the bottom-up procedure would be much more efficient since, after all, there is no way to avoid the full computation of $reach(X, Y)$ (all nodes would be reachable from 1).

To avoid this goal repetition, we have considered an alternative strategy called *tabled evaluation* [10] that can be seen as a midpoint between bottom-up and top-down. It consists in maintaining a table with previously solved goals so that they can be reused without repeating their computation. In our worst case version of the example when $source(1)$, if we compare it to the bottom-up strategy, this technique would only introduce the cost of checking the existence of each goal in the extension table.

2.1 Stratified predicates

Making a partial grounding for hidden predicates is not always applicable, as the non-computed part may affect the existence of answer sets for a given program. For instance, suppose we want to rule out solutions where we pick nodes that are reachable from 7. This is done using the constraint:

$$\leftarrow \mathit{node}(X), \mathit{in}(X), \mathit{reach}(7, X)$$

A constraint like this is usually implemented by introducing a new fresh auxiliary predicate $\mathit{aux}(X)$ and modifying the rule as follows:

$$\mathit{aux}(X) \leftarrow \mathit{node}(X), \mathit{in}(X), \mathit{reach}(7, X), \mathit{not}\ \mathit{aux}(X) \tag{5}$$

Since $\mathit{aux}(X)$ will be a hidden predicate and *is never used* in the rest of the program, a partial grounding of $\mathit{aux}(X)$ would just ignore rule (5). However, this constraint should be grounded at least for values $X = 8$ and $X = 9$ (that is, the accessible nodes from 7):

$$\begin{gathered} \mathit{aux}(8) \leftarrow \mathit{in}(8), \mathit{not}\ \mathit{aux}(8) \\ \mathit{aux}(9) \leftarrow \mathit{in}(9), \mathit{not}\ \mathit{aux}(9) \end{gathered}$$

and this should rule out all previous solutions that contained atoms $\mathit{in}(8)$ or $\mathit{in}(9)$. Furthermore, a simple rule like:

$$\mathit{aux}'(X) \leftarrow \mathit{node}(X), \mathit{not}\ \mathit{aux}'(X)$$

would make the program to have no answer sets (provided that we have at least one node) whereas partial grounding would ignore predicate $\mathit{aux}'(X)$ if it is not used in other rules.

These examples show that, in order to ignore some ground instances of a hidden predicate, we must be sure that this will not affect the set of answer sets of the program. As a simple sufficient condition, we have considered the case in which the hidden predicate is *stratified* [11]. The dependence graph of a program is formed by taking a node per each predicate and an arc $p \rightarrow q$ (meaning that *p directly depends on q*) when p is in the head of some rule in which q occurs in the body. The dependence is said to be *negative* when q is in the scope of default negation *not* , and said to be *positive* otherwise. A program is *stratified* if it contains no dependence loop that involves some negative dependence.

We say that p *depends on* a rule r, if p occurs in the head of r or there exists some path in the dependence graph from p to a predicate q occurring in the head of r. Given a program Π and a predicate p, let Π_p denote the set of rules on which p depends. A predicate p is *stratified* iff Π_p is a stratified program. Let us take now the set of stratified hidden predicates P and let Π_P the union of all Π_p for each hidden predicate $p \in P$. It is clear that we can split program Π into two parts, $\mathit{bottom}(\Pi) \stackrel{\text{def}}{=} \Pi_P$ and $\mathit{top}(\Pi) \stackrel{\text{def}}{=} \Pi \setminus \Pi_P$ that satisfy that no predicate in $\mathit{bottom}(\Pi)$ depends on rules in $\mathit{top}(\Pi)$. In this situation, we can apply the *splitting theorem* from [12] to show that the answer sets of Π can be computed in two stages: first obtain an answer set I of $bottom(\Pi)$ (in our case, I is unique) and second, simplify program $top(\Pi)$ with the information in I for predicates defined in $bottom(\Pi)$, getting answer sets for the rest of facts for remaining predicates.

In the previous example, $bottom(\Pi)$ consists of rules (1) and (2) that define predicate *reach* plus the following facts (i.e., those on which these rules depend):

$$\begin{array}{r} node(1..9).\ edge(1,2).\ edge(2,3). \\ edge(1,4).\ edge(2,5).\ edge(3,6).\ edge(4,5).\ edge(5,6). \\ edge(4,7).\ edge(5,8).\ edge(6,9).\ edge(7,8).\ edge(8,9). \end{array}$$

On the other hand, $top(\Pi) = \Pi \setminus bottom(\Pi)$ contains the rest of rules and facts in Π. Note that no atom of $bottom(\Pi)$ occurs in any rule head of $top(\Pi)$.

As a result, if part of the unique answer set I for the stratified bottom program Π_P is not actually computed, this will not affect the answer sets of the program Π provided that we actually generate the full ground program for $top(\Pi)$.

3 The experiment

As a first prototype, we have implemented an XSB Prolog program that reads an ASP program as an input. After selecting a set P of intensional hidden predicates that are stratified, we call DLV with a modification of the rest of rules $\Pi \setminus \Pi_P$ to ground all variables that can be fixed by the rest of predicates not in P. For instance, in our previous example, we would take $P = \{reach\}$ and transform rules (3)-(4) so that we capture variables that can be independently fixed without *reach* predicate:

$$\begin{array}{c} aux_1(X) \leftarrow source(X) \\ aux_2(X) \leftarrow source(X) \end{array}$$

In this way, predicate aux_1 will capture instances for variable X in (3) and aux_2 instances for X in (4). As a result, we get the partially ground program:

$$in(Y) \leftarrow reach(3,Y), not\ out(Y) \tag{6}$$

$$in(Y) \leftarrow reach(5,Y), not\ out(Y) \tag{7}$$

$$out(Y) \leftarrow reach(3,Y), not\ in(Y) \tag{8}$$

$$out(Y) \leftarrow reach(5,Y), not\ in(Y) \tag{9}$$

After this first step, we use the XSB `assert` mechanism to include the rules (1)-(2) for predicate *reach* so that any call to this predicate will be solved using the built-in tabled evaluation algorithm of XSB. Finally, for each partially grounded rule, we take non-ground atoms for hidden predicates and make the corresponding call to XSB. For instance, for (6), we make the call $reach(3, Y)$ obtaining the possible solutions $Y = 6$ and $Y = 9$ which eventually generate the ground rules:

$$in(6) \leftarrow not\ out(6)$$
$$in(9) \leftarrow not\ out(9)$$

We have performed some preliminary experiments with some random graph instances for problem in Example 1. For each random graph, we have varied the number of nodes, the *connectivity* (the number of outgoing arcs per node) and we have tried with one or two random source nodes. Results are shown in Table 1 where we compare the time obtained by a direct single call to DLV to the proposed combined execution of DLV and XSB that makes a partial grounding of hidden predicates. For graphs that are very connected, most nodes can be reached from all the rest. Thus, we can notice that the time for DLV+XSB is not significantly better or can be even worse due to additional processing and the checkings in the extension table. However, for small connectivity ratios, the time is considerably reduced. The smaller the relevant part of the graph to be traversed, the better performance obtained.

Nodes	**Connectivity**	**DLV**	**DLV+XSB**
1 source node			
500	10	0m34.581s	2m44.064s
500	20	5m14.837s	2m14.965s
500	30	9m22.146s	4m42.853s
500	40	12m20.197s	8m13.163s
500	60	17m40.125s	17m51.523s
500	100	24m2.454s	48m32.006s
2 source nodes			
600	20	4m54.504s	1m20.178s
620	20	10m32.675s	3m47.708s
640	20	12m7.537s	4m0.469s
660	20	14m3.903s	4m16.870s
700	20	16m39.363s	4m58.550s

Table 1. Running times for randomly generated graphs for problem in Example 1.

As an exaggerated case, we have constructed a graph similar to Figure 1 but of 100×100 nodes instead of 3×3, fixing 9850 as the only source node. This means that only 101 nodes are reachable from the source whereas the bottom-up computation will compute more than 10000 facts for $reach(X, Y)$ for the whole graph. DLV was able to solve the problem in 903 minutes versus DLV+XSB that took slightly more than 8 minutes.

Of course, it may be objected that the particular example we chose could have been implemented in a different way much more favorable to an efficient (complete) grounding for the problem to be considered. For instance, if we write the alternative encoding:

$$\begin{aligned} reach_s(Y) &\leftarrow source(X), edge(X,Y) \\ reach_s(Y) &\leftarrow reach_s(X), edge(X,Y) \\ in(X) &\leftarrow node(X), reach_s(X), not\ out(X) \\ out(X) &\leftarrow node(X), reach_s(X), not\ in(X) \end{aligned}$$

where this time, predicate $reach_s$ means reachable from some source node, we would obviously reduce the number of ground instances for $reach_s$ to be considered, so that partial grounding would become much less interesting. On the other hand, we cannot always expect that the program we get as an input is the best encoding for an efficient grounding. In fact, we claim that adapting each problem encoding to reduce the grounding effort usually reduces flexibility: for instance, it could be the case that rules for predicate $reach(X,Y)$ came from a general module for reachability that could be reused for other different problems.

4 Conclusions

We have considered the partial grounding of hidden predicates by treating body atoms for them as queries to be solved using tabled evaluation. We have implemented a first prototype that combines the use of an existing grounder (DLV) with the use of the XSB Prolog interpreter, which has a built-in tabled evaluation procedure. First results are promising but the application of this technique is still in a preliminary stage.

The combination of a partial grounding process plus an external tool for completing the remaining non-ground information is very similar to the technique applied in [13] where, in that case, the non-ground atoms obtained after the partial grounding correspond to constraints to be solved by a Constraint Logic Programming (CLP) tool. A relevant difference between both approaches is that, in our case, our technique does not require any syntactic or semantic extension and can be transparently applied to existing programs with the only extra requirement of specifying hidden predicates, something already present in all the existing grounders, but not currently exploited for reducing grounding.

In a future extended version of this document, we will include formal proofs for the correctness of the method. Future work also includes encoding a specialised tabled evaluation algorithm on top of the existing open source grounder `gringo`, for applying this technique on (stratified) hidden predicates.

References

1. Marek, V., Truszczyński, M. In: Stable models and an alternative logic programming paradigm. Springer-Verlag (1999) 169–181
2. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence **25** (1999) 241–273
3. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In Kowalski, R.A., Bowen, K.A., eds.: Logic Programming: Proc. of the Fifth International Conference and Symposium (Volume 2). MIT Press, Cambridge, MA (1988) 1070–1080
4. Pearce, D., Valverde, A.: Quantified equilibrium logic and foundations for answer set programs. In: Proc. of the 24th Intl. Conf. on Logic Programming (ICLP'08). (2008) 547–560
5. Ferraris, P., Lee, J., Lifschitz, V.: A new perspective on stable models. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07). (2007) 372–379
6. Niemelä, I., Simons, P.: Extending the smodels system with cardinality and weight constraints. In: Logic-Based Artificial Intelligence, Kluwer Academic Publishers (2000) 491–521
7. Denecker, M., Pelov, N., Bruynooghe, M.: Ultimate well-founded and stable semantics for logic programs with aggregates. In: 17th International Conference on Logic Programming (ICLP'01). (2001) 212–226
8. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: Proc. of the 9th European Conference on Logics in Artificial Intelligence (JELIA'04). (2004)
9. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The second Answer Set Programming competition. In: Proc. of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09). LNAI (5753), Springer-Verlag (2009) 637–654
10. Chen, W., Warren, D.S.: Tabled evaluation with delaying for general logic programs. Journal of the ACM **43**(1) (1996) 20–74
11. Lloyd, J.W.: Foundations of Logic Programming, 2nd Edition. Springer (1987)
12. Lifschitz, V., Turner, H.: Splitting a logic program. In: Proceedings of the 11th International Conference on Logic programming (ICLP'94). (1994) 23–37
13. Mellarkod, V.S., Gelfond, M., Zhang, Y.: Integrating answer set programming and constraint logic programming. Annals of Mathematics and Artificial Intelligence **53**(1-4) (2008) 251–287