DualGrounder: Lazy instantiation via Clingo multi-shot framework

Yuliya Lierler[1] and Justin Robbins[1]

University of Nebraska at Omaha, Omaha NE 68182, USA

Abstract. Answer set programming (ASP) is a declarative programming paradigm that is geared towards difficult combinatorial search problems. Sometimes, run times of ASP systems suffer due to so called grounding bottleneck. Lazy grounding solvers aim to mitigate this issue. In this paper we describe a new lazy grounding solver called DualGrounder. The DualGrounder system leverages multi-shot capabilities of the advanced ASP platform clingo. This paper also includes experimental data to explore the performance of DualGrounder compared to similar ASP grounding and solving systems.

Keywords: ASP · Lazy Grounding · Multishot Solving.

1 Introduction

Answer Set Programming (ASP) [3] is a prominent declarative programming paradigm that aims to solve difficult search problems by describing problem's specifications by means of a logic program and solving the resulting program. The process of solving ASP programs – logic programs under answer set semantics – typically involves two stages depicted in Figure 1. To describe these stages let us recall that a logic program consists of rules. When a program contains variables we call it a non-ground program, and ground otherwise. During the first stage of program's processing each rule of a logic program is converted into respective ground rules (rules without variables) in a process called *grounding*. This process involves substituting variables with all possible constant values that variables of given rules could have. The second stage is concerned with the search of so called *answer sets* (sets of ground atoms representing solutions) of the constructed ground program. The basic way of performing this process is to (i) utilize a grounder, for instance, the gringo system [10] to ground an ASP program, and then (ii) pipe the output to a solving system such as clasp [11]

Fig. 1. Typical ASP system architecture.

or WASP [2]. However, grounding some programs may prove to be a bottleneck in applying ASP technology. Converting a rule with variables into rules with respective constants may require, in the worst case scenario, a substitution of every possible combination of program's constants into the rule. "*Lazy ASP*" methods such as implemented in systems ASPERIX [12], GASP [7], and OMIGA [8] combat this issue by altering the typical ground-solve architecture of ASP systems described here. Lazy ASP architectures delay grounding of some parts of a program until it is determined to be necessary. At times grounding these parts is never necessary. In this work, we design a lazy ASP architecture and implement it within system DUALGROUNDER, or DG, for short. It's a close relative of the WASP-based lazy ASP systems proposed and advocated in [5, 6]. The novelty of the DG tool is its reliance on the CLINGO multi-shot framework [9]. This way in place of implementing in house procedures for grounding and solving in various stages of lazy approach we rely on existing instances of GRINGO and CLASP withing the CLINGO multi-shot architecture. Here we describe the exact architecture of the DG system and provide the experimental analysis of the approach comparing it to its close relatives.

2 Preliminaries

A *logic/ASP program* is a finite set of rules of the form

$$h_1 \mid \cdots \mid h_n \leftarrow a_1, \ldots a_l,\ \mathit{not}\ a_{l+1}, \ldots,\ \mathit{not}\ a_m, \tag{1}$$

where $m, n \geq 0$, h_i $(1 \leq i \leq n)$ and $a_1, ...a_m$ are atoms. Expressions to the left hand side of an arrow and the right hand side of an arrow are called *head* and *body* of a rule, respectively. An atom, literal, or rule is *ground* if it has no variables within terms occurring in it. We call a rule a *fact* if its body is empty $(m = 0)$. A rule is a *constraint* if its head is empty $(n = 0)$; in this case we can identify it with the symbol $\bot$. We say that a rule is *disjunctive* if its head has multiple atoms $(n > 1)$. Intuitively, constraints are meant to capture a condition – by means of a set of literals – that should *not* take place in a valid solution to the problem. We assume that a reader is familiar with the definition of an answer set of a (ground) logic program and refer to the paper by Lifschitz et al. [13] for details. One crucial result that constitutes a theoretical basis for this work is the following theorem by Lifschitz et al. [13]:

Theorem 1 (Theorem on Constraints). *For a ground program P and a set C of constraints so that $C \subseteq P$, a set X of atoms is an answer set of P iff X is an answer set of program $P \setminus C$ and X satisfies every constraint in C.*

It tells us that when a program contains constraints we may split a task of computing its answer sets into two subtasks. In the first subtask, we are concerned with finding answer sets of a program resulting from the original program some of whose constraints are removed. In the second subtask, we are concerned with checking that these constraints are satisfied. Grounding a logic program with

variables replaces each rule with all its instances obtained by substituting the object constants occurring in the program, for all variables. For a program P, by $ground(P)$ we denote the result of its grounding. *Answer sets* of a logic program P with variables are the answer sets of ground program $ground(P)$. It is easy to see how the theorem on constraints generalizes to the case of programs with variables.

In the introduction we presented a common architecture for processing logic programs. Yet, modern answer set solvers are complex software systems that are designed to accommodate a number of potential uses that go beyond their typical utilization. For example, such answer set solvers as CLASP and WASP allow for something that we will denote *incremental solving*. Incremental ASP-solving allows the user to solve several ground logic programs $P_1, P_2, \ldots, P_n$ one after another (possibly in an "online" mode when an instance of a solver is never terminated but rather is put "on-hold" while preserving its internal search state), if P_{i+1} results from P_i by adding ground constraints. In this case the search for a solution to P_{i+1} may benefit from the knowledge obtained during solving $P_1, \ldots, P_i$ sequence.

3 System DG

Lazy Instantiation by Cuteri et al. [5, 6] Here we review a lazy instantiation method for finding answer sets of a program, studied by Cuteri et al. [5]. The method separates a program P into a program composed of a predetermined subset of its constraints C, and a program composed of the remaining rules $P_{nc} = P \setminus C$. Program P_{nc} is processed using the typical ground and solve process depicted in Figure 1, except on the onset of solving an instance of an ASP solver capable of incremental solving is considered. As a result an answer set M of P_{nc} is computed. This answer set M is then checked against the constraints in C. If all of the ground instances of the constraints in C are satisfied by M, then M is returned by the method as an answer set of P. Otherwise, ground instances of constraints that are violated by the candidate model are provided to an incremental ASP solver to proceed with the search. The process is repeated up to the point when we are able to either claim that a found M is indeed an answer set of P or establish that P has no answer sets. It is easy to see that at any point of computation an incremental solver is dealing with some subset of $ground(P)$. There are two interesting peculiarities of the approach studied by Cuteri et al. [5]. First, the process of checking current answer set M of some subset of $ground(P)$ against the appropriate set of constraints stemming from C is a custom program produced automatically for each unique problem. In particular, the authors illustrated the case study on three benchmarks. The authors implemented such a check individually for each benchmark via a specialized propagator interface provided by such answer set solvers as CLASP and WASP. Second, the process of computing ground instances of constraints at hand to extend incrementally solved program was once more implemented by a custom program designed for each problem. Cuteri et al. [5] demonstrated positive results

for their case study. Cuteri et al. [6] make the approach described above problem/benchmark independent. They developed an answer set solver based on lazy instantiation method described here that is problem agnostic. In other words, their method is able to utilize constraints of C to implement "a propagator" and then "a grounder" for these constraints to communicate with an incremental solver at hand. C++ is used to implement procedures of above mentioned propagator and grounder based on the information provided by constraints in C.

DG *specifics* The DG system mimics the efforts by Cuteri et al. [6]. The key difference in our undertaking is the utilization of the available off the shelf tools for the task of grounding (in particular, GRINGO) rather than implementing a custom solution for this purpose. Implementation of system DG relies on the CLINGO multi-shot framework [9] in a way that the key computational tasks are executed by instances of grounding and solving routines available via this framework. Thus, the role of DG is to orchestrate these routines. First, system DG separates given program P into two parts: a program composed of a specified subset of its constraints C, and a program composed of the remaining rules $P_{nc} = P \setminus C$. Second, system DG rewrites constraints in C procedurally. We use an example to illustrate this rewriting. Assume a sample constraint `:- p1(X), p2(Y), not p3(X,Y).` Rule `p1'1_p2'2_not'p3'12'(X,Y):- p1(X), p2(Y), not p3(X,Y).` is used in place of this constraint. Constraints in C rewritten in this way result in program C'. Third, DG orchestrates the back-and-forth communication of two major subroutines that we call MAIN-GS and AUX-G (program C' plays a crucial role in the workings of the AUX-G component).

MAIN-GS**:** A grounder-solver pair MAIN-GS is responsible for incremental solving procedure of DG. It is first applied to P_{nc} to compute one answer set M ; this answer set is then used within the second subroutine AUX-G to either (i) establish that this set M of atoms is indeed an answer set of P or (ii) compute ground constraints due to C that are violated by M; these constraints are then added incrementally to the logic program of MAIN-GS and a solver of MAIN-GS is instructed to find a new answer set to repeat the described process.

AUX-G**:** The AUX-G routine is responsible for supplying ground instances of constraints in C violated by the "candidate" answer set M given by MAIN-GS. Each time AUX-G subroutine is called it uses a new instance of a grounder GRINGO supplying it with a new program to ground. Component AUX-G calls grounder GRINGO on program $C' \cup M$ (here we identify set M of atoms with the set of facts constructed from its elements). Due to the inner workings of GRINGO and structure of program $C' \cup M$, GRINGO's output consists of M together with the facts such as `p1'1_p2'2_not'p3'12'(4,5)` (following our earlier example). Facts of the form `p1'1_p2'2_not'p3'12'(4,5)` are translated procedurally by system DG into constraints of the form `:- p1(4), p2(5), not p3(4,5).` Such constraints are added incrementally to the MAIN-GS grounder-solver pair of DG. If given some candidate answer set M, GRINGO invoked on $C' \cup M$ returns M itself, the DG system returns M to the user as it is indeed the answer set to the given program P as no constraints in C are violated.

4 Experimental Evaluation

Our experiments were run on a Linux machine, where each instance's runtime was limited to 10 minutes and given 16GB of memory to work with. The benchmark called *Packing* was given an extended 30 minute cutoff. Table 1 summarizes the outcomes of our experimental analysis. The dualgrounder implementation used for the experiments can be found at *https://www.unomaha.edu/college-of-information-science-and-technology/natural-language-processing-and-knowledge-representation-lab/software/dualgrunder.php*. We now provide the details on considered systems and benchmarks. In parenthesis we give abbreviations used in Table 1. We tested two variants of DG, one with default CLINGO settings (*DG-Clingo*) and another with settings/flags that emulate the heuristics of the WASP solver (*DG-Wasp*) This was done to enable better comparison with other systems used in our experiments. We provide run times of systems CLINGO, WASP, CLINGO with a lazy propagator (*Clingo-Lazy*) [5], WASP with a lazy propagator (*Wasp-Lazy*) [5], and the partial compilation system (*Partial-Comp*) [6]. We used three benchmarks to assess performance of the DG system: *Stable Marriage*, *Natural Language Understanding*, and *Packing*. These benchmarks come from the experimental analysis by Cuteri et al. [5, 6]. We refer the reader to these papers for exhaustive details about these benchmarks; the constraints selected for lazy grounding mirror those chosen in these papers. Here we only include few remarks on these problems.

Stable Marriage (SM). Our experiments utilize the 2013 encoding of Stable Marriage from the fourth ASP competition [1]. The lazily grounded constraint for SM checks to see if a couple would rather be with someone else than each other:

```
:- match(M,W1), manAssignsScore(M,W,Smw), W1 != W, , Smw > Smw1,
   manAssignsScore(M,W1,Smw1) match(M1,W),Swm >= Swm1,
   womanAssignsScore(W,M,Swm), womanAssignsScore(W,M1,Swm1).
```

Natural Language Understanding (NLU). First introduced by Schüller [14], the NLU benchmark determines the solution to first order horn clause abduction problems under a variety of cost functions. These cost functions are the cardinality minimality (*card*), cohesion (*coh*), and weight abduction (*wa*) functions. The lazily grounded constraint for these problems ensures transitivity between equation atoms: `:- eq(A,B), eq(B,C), not eq(A,C).`

Packing. The third benchmark in the DG experiments is the packing problem, in which the goal of the problem is to pack a number of different squares into a rectangular area such that none of their areas overlap. This problem is the same as the packing problem in the third ASP competition [4]. The constraints that check for overlap between each defined square and those that check square positions are difficult to ground, so they are lazily evaluated:

```
:- pos(I,X,Y), pos(I,X1,Y1), X1 != X.
:- pos(I,X,Y), pos(I,X1,Y1), Y1 != Y.
:- pos(I1,X1,Y1), square(I1,D1), pos(I2,X2,Y2), square(I2,D2),
   I1!=I2, W1=X1+D1, H1=Y1+D1, X2>=X1, X2<W1, Y2>=Y1, Y2<H1.
```

Table 1. Experimental test results; the average execution time is displayed along with the number of timed out instances for each system and benchmark.

	SM(210)	**card(50)**	**coh(50)**	**wa(50)**	**Packing(50)**
Clingo	**229.307 (26)**	74.613 (5)	67.303 (7)	78.069 (7)	1521.853 (49)
Clingo-Lazy	142.992 (91)	5.761 (0)	6.211 (0)	6.472 (0)	556.457 (31)
DG-Clingo	93.168 (80)	**2.197 (0)**	2.718 (0)	3.193 (0)	415.834 (38)
DG-Wasp	94.814 (116)	2.374 (0)	**2.598 (0)**	**2.785 (0)**	417.251 (38)
Wasp	186.613 (55)	111.309 (3)	112.397 (3)	135.637 (2)	1550.349 (46)
Partial-Comp	27.613 (64)	5.049 (0)	15.311 (2)	51.757 (1)	**447.513 (28)**
Wasp-Lazy	11.606 (68)	3.078 (0)	23.292 (1)	37.657 (1)	306.962 (38)

Results Discussion. The goal of our experimentation was to evaluate the performance of the DG system. Table 1 presents the experimental outcomes. The number following the benchmark names is the total number of the instances considered. The numbers in columns are average run times of the systems on instances that did not timeout. The number in parenthesis specifies number of timed out instances.

For the NLU tests (columns card, coh, wa), DG tends to perform slightly better than other lazy grounding systems, while systems that did not utilize lazy grounding fell behind by a large margin. The NLU benchmark seems to benefit greatly from the removal of its "problem" rules. DG's slightly higher performance when compared to other lazy grounding systems seems to mostly stem from the lack of overhead in DG when a solution is produced on the first cycle; if a solution is found, no constraints are constructed and the program ends. In contrast to the NLU benchmark, DG's performance on the Stable Marriage and Packing benchmarks drops significantly compared to the other tested systems, especially on SM. We believe this is due to the string processing done by the system during constraint construction, and by the increased overhead caused by using Python over the C++ code used by, for example, the Part-Comp approach. For Stable Marriage, this is compounded by the fact that the problem is not a good fit for the lazy grounding approach, as all of the lazy grounding systems hit slower execution times than the base CLINGO or WASP systems. The Packing problem presents a problem that is very difficult for both the base systems and the lazy grounding systems; none of the tested systems were able to solve all of the tested instances. The performance of DG seems comparable to that of its lazy grounding peers that overall outperform CLINGO and WASP.

Conclusions and Acknowledgements We trust that the DG system is a valuable representative among the class of lazy grounding systems. We also see its great value in showcasing how CLINGO multi-shot framework can be used in apparently "unintended" and meaningful ways. The work was partially supported by NSF grant 1707371.

References

1. Alviano, M., Calimeri, F., Charwat, G., Dao-Tran, M., Dodaro, C., Ianni, G., Krennwallner, T., Kronegger, M., Oetsch, J., Pfandler, A., Pührer, J., Redl, C., Ricca, F., Schneider, P., Schwengerer, M., Spendier, L.K., Wallner, J.P., Xiao, G.: The fourth answer set programming competition: Preliminary report. In: Cabalar, P., Son, T.C. (eds.) Logic Programming and Nonmonotonic Reasoning. pp. 42–53. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
2. Alviano, M., Dodaro, C., Faber, W., Leone, N., Ricca, F.: WASP: A native ASP solver based on constraint learning. In: Proc. of LPNMR 2013. LNCS, vol. 8148, pp. 54–66. Springer-Verlag (2013)
3. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. Communications of the ACM **54(12)**, 92–103 (2011)
4. Calimeri, F., Ianni, G., Ricca, F.: The third open answer set programming competition. Theory and Practice of Logic Programming **14**(1), 117–135 (2014). https://doi.org/10.1017/S1471068412000105
5. Cuteri, B., Dodaro, C., Ricca, F., Schüller, P.: Constraints, lazy constraints, or propagators in ASP solving: An empirical analysis. Theory and Practice of Logic Programming **17**(5-6), 780–799 (2017)
6. Cuteri, B., Dodaro, C., Ricca, F., Schüller, P.: Partial Compilation of ASP Programs. Theory and Practice of Logic Programming **19**(5-6), 857–873 (2019). https://doi.org/10.1017/S1471068419000231
7. Dal Palù, A., Dovier, A., Pontelli, E., Rossi, G.: GASP: Answer set programming with lazy grounding. Fundamenta Informaticae (2009). https://doi.org/10.3233/FI-2009-180
8. Dao-Tran, M., Eiter, T., Fink, M., Weidinger, G., Weinzierl, A.: OMiGA: An open minded grounding on-the-fly answer set solver. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (2012)
9. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Multi-shot asp solving with clingo. Theory and Practice of Logic Programming **19**(1), 27–82 (2019). https://doi.org/10.1017/S1471068418000054
10. Gebser, M., Kaminski, R., König, A., Schaub, T.: Advances in gringo series 3. In: Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR). pp. 345–351. Springer (2011). https://doi.org/10.1007/978-3-642-20895-9_39, `http://dx.doi.org/10.1007/978-3-642-20895-9\_39`
11. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artificial Intelligence **187**, 52–89 (2012)
12. Lefèvre, C., Nicolas, P.: The First Version of a New ASP Solver : ASPeRiX (2009). https://doi.org/10.1007/978-3-642-04238-6_52
13. Lifschitz, V., Tang, L.R., Turner, H.: Nested expressions in logic programs. Annals of Mathematics and Artificial Intelligence **25**, 369–389 (1999)
14. Schüller, P.: Modeling variations of first-order horn abduction in answer set programming. Fundamenta Informaticae **149**(1-2), 159–207 (2016). https://doi.org/10.3233/FI-2016-1446