# On the Automated Selection of ASP Instantiators*

Marco Maratea[1], Luca Pulina[2], and Francesco Ricca[3]

[1] DIBRIS, Univ. degli Studi di Genova, Viale F. Causa 15, 16145 Genova, Italy
marco@dist.unige.it
[2] POLCOMING, Univ. degli Studi di Sassari, Viale Mancini 5, 07100 Sassari, Italy
lpulina@uniss.it
[3] Dip. di Matematica ed Informatica, Univ. della Calabria, Via P. Bucci, 87030 Rende, Italy,
ricca@mat.unical.it

**Abstract.** Answer Set Programming (ASP) is a powerful language for knowledge representation and reasoning. ASP is exploited in real-world applications and is also attracting the interest of industry thanks to the availability of efficient implementations. ASP systems compute solutions relying on two modules: an *instantiator* (or *grounder*) that produces, by removing variables from the rules, a ground program equivalent to the input one; and a *model generator* (or *solver*) that computes the solutions of such propositional program. In this paper we make a first step toward the exploitation of automated selection techniques to the grounding module. We rely on two well-known ASP grounders, namely the grounder of the DLV system and GRINGO, and we leverage on automated classification algorithms to devise and implement an automatic procedure for selecting the "best" grounder for each problem instance. An experimental analysis, conducted on benchmarks and solvers from the 3rd ASP Competition, shows that our approach improves the evaluation performance independently from the solver associated with our grounder selector.

## 1 Introduction

In the last decade, Answer Set Programming (ASP) [?,?,?,?,?,?] is increasingly attracting the interest of industry thanks to the availability of efficient implementations – see, e.g., [?]. Thus, the ability of ASP systems to compute solutions in an efficient way is a factor of paramount importance, especially when they deal with industrial level problems. Given a non-ground ASP program, ASP systems compute solutions relying on two main modules: an *instantiator* (or *grounder*) that produces a propositional ASP program, and a *model generator* (or *solver*) that takes as input a propositional program and returns a solution.

Recently, the application of automated algorithm selection techniques to ASP solving [?,?,?,?,?] has noticeably improved the performance of ASP systems. The approaches cited above are often obtained by importing to ASP techniques already applied to Constraint Satisfaction problems, propositional satisfiability (SAT) or Quantified SAT (see [?,?,?] for details). The drawback about the adoption of such techniques

* This paper is an early version of the work submitted at the 13rd Conference of the Italian Association for Artificial Intelligence (AI*IA 2013).

is that they are confined to the model generator module, mainly because the research fields mentioned deal with inherently ground problems.

In ASP it is well-established that limiting the choice to only one grounder could avoid the exploitation of several optimization techniques, possibly applied to different problem class (e.g., Polynomial (*P*) and *NP* problem classes as classified in the 3rd ASP Competition [**?**]), implemented only in one grounder (e.g. Magic sets [**?**] in the presence of queries).

In this paper we make a first step toward the exploitation of automated selection techniques to the grounding module. We rely on two well-known ASP grounders, namely the grounder of the DLV system [**?**] (DLV-G in the following) and GRINGO [**?**], and we leverage on automated classification algorithms to automatically select the "best" grounder. More in details, our starting point is an experimental analysis conducted on the domains of benchmarks belonging to both *P* and *NP* classes of the 3rd ASP Competition, involving state-of-the-art ASP solvers and the aforementioned grounders. We then applied classification methods by relying on characteristics of the various encoding (*features*), with the aim of automatically select the most appropriate grounder. We then implemented a system based on these ideas, and the results of our experimental analysis show that the performance of the considered solvers are boosted by the usage of the proposed system.

To sum up, the main contributions of this paper are:

- the application of automated selection techniques to the grounding module in ASP computation, to complement a recent body of research only focused on solvers;
- the implementation of a system based on these techniques; and
- an experimental analysis of the new system, involving a large variety of benchmarks, grounders and solvers, that shows the benefits of the approach.

The paper is structured as follows. Section 2 introduces needed preliminaries about ASP. Section 3 shows the main answer set computation methods, with focus on the grounding module. Section 4 then describes the ideas we have applied to reach the above-mentioned goals. Section 5 presents implementation details of the system implemented along the line reported in the previous section, and its results. The paper ends with some conclusions in Section 6.

## 2 Answer Set Programming

In this section we recall Answer Set Programming syntax and semantics.

**Syntax.** A variable or a constant is a *term*. An *atom* is $p(t_1, ..., t_n)$, where $p$ is a *predicate* of arity $n$ and $t_1, ..., t_n$ are terms. A *literal* is either a *positive literal* $p$ or a *negative literal* $\texttt{not}\ p$, where $p$ is an atom. A *(disjunctive) rule* $r$ has the following form:

$$a_1 \lor \ldots \lor a_n \text{ :- } b_1, \ldots, b_k, \texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_m. \quad (1)$$

where $a_1, \ldots, a_n, b_1, \ldots, b_m$ are atoms. The disjunction $a_1 \lor \ldots \lor a_n$ is the *head* of $r$, while the conjunction $b_1, \ldots, b_k, \texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_m$ is the *body* of $r$. A rule having

precisely one head literal (i.e. $n = 1$) is called a *normal rule*. If the body is empty (i.e. $k = m = 0$), it is called a *fact*, and the $:-$ sign is usually omitted. A rule without head literals (i.e. $n = 0$) is usually referred to as an *integrity constraint*. A rule $r$ is *safe* if each variable appearing in $r$ appears also in some positive body literal of $r$.

An *ASP program* $\mathcal{P}$ is a finite set of safe rules. A `not` -free (resp., $\vee$-free) program is called *positive* (resp., *normal*). A term, an atom, a literal, a rule, or a program is *ground* if no variables appear in it.

Hereafter, we denote by $H(r)$ the set $\{a_1, \ldots, a_n\}$ of the head atoms, and by $B(r)$ the set $\{b_1, \ldots, b_k, \texttt{not } b_{k+1}, \ldots, \texttt{not } b_m\}$ of the body literals. $B^+(r)$ (resp., $B^-(r)$) denotes the set of atoms occurring positively (resp., negatively) in $B(r)$. A predicate $p$ is referred to as an *EDB* predicate if, for each rule $r$ having in the head an atom whose name is $p \in H(r)$, $r$ is a fact; all others predicates are referred to as *IDB* predicates. The set of facts in which *EDB* predicates occur, denoted by $EDB(\mathcal{P})$, is called *Extensional Database (EDB)*, the set of all other rules is the *Intensional Database (IDB)*.

*Semantics.* Let $\mathcal{P}$ be an ASP program. The *Herbrand universe* of $\mathcal{P}$, denoted as $U_{\mathcal{P}}$, is the set of all constants appearing in $\mathcal{P}$. In the case when no constant appears in $\mathcal{P}$, an arbitrary constant is added to $U_{\mathcal{P}}$. The *Herbrand base* of $\mathcal{P}$, denoted as $B_{\mathcal{P}}$, is the set of all ground atoms constructable from the predicate symbols appearing in $\mathcal{P}$ and the constants of $U_{\mathcal{P}}$. Given a rule $r$ occurring in a program $\mathcal{P}$, a *ground instance* of $r$ is a rule obtained from $r$ by replacing every variable $X$ in $r$ by $\sigma(X)$, where $\sigma$ is a substitution mapping the variables occurring in $r$ to constants in $U_{\mathcal{P}}$. We denote by $Ground(\mathcal{P})$ the set of all the ground instances of the rules occurring in $\mathcal{P}$.

An *interpretation* for $\mathcal{P}$ is a set of ground atoms, that is, an interpretation is a subset $I$ of $B_{\mathcal{P}}$. A ground positive literal $A$ is true (resp., false) w.r.t. $I$ if $A \in I$ (resp., $A \notin I$). A ground negative literal `not` $A$ is true w.r.t. $I$ if $A$ is false w.r.t. $I$; otherwise `not` $A$ is false w.r.t. $I$. Let $r$ be a rule in $Ground(\mathcal{P})$. The head of $r$ is true w.r.t. $I$ if $H(r) \cap I \neq \emptyset$. The body of $r$ is true w.r.t. $I$ if all body literals of $r$ are true w.r.t. $I$ (i.e., $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$) and otherwise the body of $r$ is false w.r.t. $I$. The rule $r$ is *satisfied* (or true) w.r.t. $I$ if its head is true w.r.t. $I$ or its body is false w.r.t. $I$. A *model* for $\mathcal{P}$ is an interpretation $M$ for $\mathcal{P}$ such that every rule $r \in Ground(\mathcal{P})$ is true w.r.t. $M$. A model $M$ for $\mathcal{P}$ is *minimal* if there is no model $N$ for $\mathcal{P}$ such that $N$ is a proper subset of $M$. The set of all minimal models for $\mathcal{P}$ is denoted by $\mathrm{MM}(\mathcal{P})$. In the following, the semantics of ground programs is first given, then the semantics of general programs is given in terms of the answer sets of its instantiation. Given a *ground* program $\mathcal{P}$ and an interpretation $I$, the *reduct* of $\mathcal{P}$ w.r.t. $I$ is the subset $\mathcal{P}^I$ of $\mathcal{P}$ obtained by deleting from $\mathcal{P}$ the rules in which a body literal is false w.r.t. $I$.[4] Let $I$ be an interpretation for a ground program $\mathcal{P}$. $I$ is an *answer set* (or stable model) for $\mathcal{P}$ if $I \in \mathrm{MM}(\mathcal{P}^I)$ (i.e., $I$ is a minimal model for the program $\mathcal{P}^I$) [**?**].

*Queries.* A program $\mathcal{P}$ can be coupled with a *query* in the form $q?$, where $q$ is a literal. Let $\mathcal{P}$ be a program and $q?$ be a query, $q?$ is *true* iff for any answer set $A$ of $\mathcal{P}$ it holds that $q \in A$. Basically, the semantics of queries corresponds to cautious reasoning, since a query is true if the corresponding atom is true in all answer sets of $P$.

[4] This definition, introduced in [**?**], is equivalent to the one of Gelfond and Lifschitz [**?**].

## 3 Answer Sets Computation

In this section we overview the evaluation of ASP programs, and recall the available solutions mentioning the techniques underlying the state of the art implementations.

The evaluation of ASP programs is traditionally carried out in two phases: program instantiation and model generation. As a consequence, an ASP system usually couples two modules: the *grounder* or *instantiator* and the *ASP model generator* or *solver*. In the following we provide a more detailed description of the instantiation, since the target of this work is improving performance of this phase.

*ASP Program Instantiation.* In general, an ASP program $\mathcal{P}$ contains variables, and the process of *instantiation or grounding* aims to eliminate these variables in order to generate a propositional ASP program equivalent to $\mathcal{P}$. Note that, the full theoretical instantiation $Ground(\mathcal{P})$ introduced in previous section contains all the ground rules that can be generated applying every possible substitution of variables. A modern instantiator module does not produce the full ground instantiation $Ground(\mathcal{P})$ (which is unnecessarily huge in size), but employees several techniques to produce one that is both equivalent and usually much smaller than $Ground(\mathcal{P})$. Notice that grounding is an EXPTIME-hard task, indeed in general it may produce a program that is of exponential size w.r.t. the input program. Thus, having an instantiator able to produce a comparatively small program in a reasonable time is crucial to achieve good (or even acceptable) performance in evaluating ASP programs. For instance, DLV-G generates a ground instantiation that has the same answer sets as the full one, but is much smaller in general [**?**].

In order to generate a small ground program equivalent to $\mathcal{P}$, a modern instantiator usually exploits some structural information on the input program. The evaluation proceeds bottom-up, starting from the information contained in the facts and evaluating the rules according to the positive body-to-head dependencies. Such dependencies can be identified by mean of the *Dependency Graph* (DG) of $\mathcal{P}$. The DG of $\mathcal{P}$ is a directed graph $G(\mathcal{P}) = \langle N, E \rangle$, where $N$ is a set of nodes and $E$ is a set of arcs. $N$ contains a node for each IDB predicate of $\mathcal{P}$, and $E$ contains an arc $e = (p, q)$ if there is a rule $r$ in $\mathcal{P}$ such that $q$ occurs in the head of $r$ and $p$ occurs in a positive literal of the body of $r$. The graph $G(\mathcal{P})$ induces a subdivision of $\mathcal{P}$ into subprograms (also called *modules*) allowing for a modular evaluation. We say that a rule $r \in \mathcal{P}$ *defines* a predicate $p$ if $p$ appears in the head of $r$. For each strongly connected component (SCC) $C$ of $G(\mathcal{P})$, the set of rules defining all the predicates in $C$ is called *module* of $C$ and is denoted by $\mathcal{P}_c$.

The DG induces a partial ordering among its SCCs which is followed during the evaluation. Basically, this order allows to perform a layered evaluation of the program; one module at time in such a way that data needed for the instantiation of a module $C_i$ have been already generated by the instantiation of the modules preceding $C_i$. This way, ground instances of rules are generated using only atoms which can possibly be derived from $\mathcal{P}$, and thus avoiding the combinatorial explosion that may occur in the case of a full instantiation [**?**]. Modules containing recursive rules are evaluated according to fix-point techniques originally introduced in the field of deductive databases [**?**].

In turn, each rule in a module is processed by applying a variable to constant matching procedure, which is basically implemented as a backtracking algorithm. Actually, modern instantiators implement a backjumping technique [?]. Note that the instantiation of a rule is very similar to the evaluation of a conjunctive query, which is a process exponential both in the size of the query (number of elements in the body) and in the number of variables. An additional aspect influencing the cost of evaluating a rule, is the way variables are bound (trough joins or builtin operators), as it also happens for conjunctive queries [?].

At the time of this writing, two are the most prominent instantiators for ASP programs, which are capable of parsing the core language employed in the 3rd ASP competition, namely DLV-G and GRINGO. These two grounders are both based on the above-mentioned techniques, but employ specific variants and heuristics which are described in the related literature [?,?,?], and that will be outlined in Section 5 when the results are analyzed.

*Answer Set Solving.* The subsequent computations, which constitute the non-deterministic part of ASP programs evaluation, are then performed on the ground instantiation by an *ASP solver*. ASP solvers employ algorithms very similar to SAT solvers, i.e., specialized variants of DPLL [?] search.

There are several different approaches to ASP solving that range from native solvers (i.e., implementing ASP-specific techniques), to rewriting-based solutions (e.g., rewriting programs into SAT and calling a SAT solver). Among the ones that participated to the 3rd ASP Competition, we recall the native ASP solvers SMODELS [?], DLV [?] and CLASP [?]. SMODELS is one the first ASP systems made available; and DLV is one of the first robust implementations able to cope with disjunctive programs. Both feature look-ahead based techniques and ASP-specific search space pruning operators. CLASP is a native ASP solver relying on conflict-driven nogood learning. Among the rewriting-based ASP solvers we mention CMODELS [?], IDP [?], and the LP2SAT [?] family that resort on a translation to SAT. There are also proposals, like the LP2DIFF [?] family, rewriting ASP in difference logic and calling a Satisfiability Modulo Theories solver to compute answer sets.

## 4 Automated selection of grounding algorithm

In our previous work [?,?], our aim was to build an efficient ASP solver on top of state-of-the-art systems, leveraging on machine learning techniques to automatically choose the "best" available solver on a per-instance basis. Our analysis focused on *ground* instances, and, to do that, we ran each non-ground instance with GRINGO – the same setting used in the 3rd ASP Competition –, letting our system ME-ASP to choose the best solver to fire. In order to extend ME-ASP to cope with non-ground instances, and considering that in [?] we report that ME-ASP was not able to cope with a number of instances due to GRINGO failures during the grounding stage, to obtain a more efficient system we investigate the application of algorithm selection techniques to the grounding phase, by relying on DLV-G and GRINGO.

Considering the grounders described above, it is not clear which one represents the choice that allows to reach the best possible performance. In the context of the

| Problem | Class | Problem | Class |
|---|---|---|---|
| DisjunctiveScheduling | *NP* | HydraulicLeaking | *P* |
| HydraulicPlanning | *P* | GrammarBasedIE | *P* |
| GraphColouring | *NP* | HanoiTower | *NP* |
| KnightTour | *NP* | MazeGeneration | *NP* |
| Labyrinth | *NP* | MCSQuerying | *NP* |
| Numberlink | *NP* | PackingProblem | *NP* |
| PartnerUnitsPolynomial | *P* | Reachability | *P* |
| SokobanDecision | *NP* | Solitaire | *NP* |
| StableMarriage | *P* | WeightAssignmentTree | *NP* |

**Table 1.** Pool of ASP problems involved in the reported experiments. Notice that "GrammarBasedIE" and "MCSQuerying" are shorthands for the problems named "GrammarBasedInformationExtraction" and "MultiContextSystemQuerying", respectively.

3rd ASP Competition, GRINGO has been used as grounder for all participant solvers, mainly because it features an easy numeric format (i.e. the one of LPARSE [?]), that all participant solvers can read and employ.

In order to investigate this point, we design an experiment aimed to highlight the performance of a pool of state-of-the-art ASP solvers on a pool of problem instances. Concerning the solvers, we selected the pool comprised in the multi-engine solver ME-ASP, namely CLASP, CMODELS, DLV and IDP. As reported in [?][5], these solvers are representative of the state-of-the-art solver (SOTA), i.e., considering a problem instance, the oracle that always fares the best among available solvers.

The benchmarks considered for the experiment belong to the suite of the 3rd ASP Competition. This is a large and heterogeneous suite of benchmarks encoded in ASP-Core, which was already employed for evaluating the performance of state-of-the-art ASP solvers. That suite includes planning domains, temporal and spatial scheduling problems, combinatorial puzzles, graph problems, and a number of application domains, i.e., database, information extraction and molecular biology field [6]. In more detail, we have employed the encodings used in the System Track of the competition of all evaluated problems belonging to the categories *P* and *NP*, and all the problem instances evaluated at the competition[7]. Notice that with *instance* we refer to the complete input program (i.e., encoding+facts) to be fed to a solver for each instance of the problem to be solved. In Table 1 we report the problems involved in our experiment. We evaluated 10 instances per problem – the same ones evaluated at the System Track of the 3rd ASP Competition –, for a total amount of 180 instances.

In Table 2 we report the results of the experiment described above. All the experiments ran on a cluster of Intel Xeon E31245 PCs at 3.30 GHz equipped with 64 bit Ubuntu 12.04, granting 600 seconds of CPU time for the whole process (grounding + solving) and 2GB of memory to each system. We present Table 2 in two parts – top

[5] A preliminary version is available for download at [?].
[6] An exhaustive description of the benchmark problems can be found in [?].
[7] Both encodings and problem instances are available at the competition website [?].

| | CLASP | | | | CMODELS | | | |
|---|---|---|---|---|---|---|---|---|
| | DLV-G | | GRINGO | | DLV-G | | GRINGO | |
| | # | Time | # | Time | # | Time | # | Time |
| DisjunctiveScheduling | 10 | 425.20 | 5 | 75.45 | 9 | 977.82 | 4 | 947.17 |
| GrammarBasedIE | 10 | 2323.72 | 10 | 254.33 | 10 | 2344.88 | 10 | 266.69 |
| GraphColouring | 3 | 20.90 | 3 | 144.55 | 4 | 423.54 | 4 | 357.90 |
| HanoiTower | 6 | 751.65 | 7 | 1058.87 | 7 | 314.03 | 7 | 137.11 |
| HydraulicLeaking | 10 | 2095.85 | 7 | 2819.15 | 10 | 2087.08 | 7 | 2850.30 |
| HydraulicPlanning | 10 | 883.17 | 10 | 154.30 | 10 | 878.80 | 10 | 164.57 |
| KnightTour | 7 | 148.76 | 7 | 82.52 | 6 | 71.19 | 6 | 18.50 |
| Labyrinth | 9 | 720.53 | 10 | 237.50 | 8 | 399.77 | 9 | 580.40 |
| MazeGeneration | 10 | 35.54 | 10 | 4.94 | 10 | 120.03 | 10 | 5.94 |
| MCSQuerying | 10 | 136.38 | 10 | 130.82 | 10 | 155.84 | 10 | 133.55 |
| Numberlink | 8 | 254.06 | 7 | 9.64 | 4 | 161.19 | 4 | 353.41 |
| PackingProblem | 9 | 2285.40 | – | – | 9 | 2247.19 | – | – |
| PartnerUnitsPolynomial | 8 | 263.70 | 2 | 63.94 | 8 | 262.93 | – | – |
| Reachability | 9 | 528.69 | 6 | 110.50 | 8 | 427.97 | 5 | 439.69 |
| SokobanDecision | 10 | 425.09 | 10 | 512.59 | 10 | 814.07 | 10 | 893.03 |
| Solitaire | 3 | 206.73 | 2 | 81.51 | 4 | 303.09 | 3 | 488.84 |
| StableMarriage | 1 | 61.47 | – | – | – | – | – | – |
| WeightAssignmentTree | 8 | 416.69 | 1 | 15.62 | 5 | 1100.25 | – | – |

| | DLV | | | | IDP | | | |
|---|---|---|---|---|---|---|---|---|
| | DLV-G | | GRINGO | | DLV-G | | GRINGO | |
| | # | Time | # | Time | # | Time | # | Time |
| DisjunctiveScheduling | 1 | 35.09 | 1 | 44.60 | 10 | 415.90 | 5 | 69.36 |
| GrammarBasedIE | 10 | 2020.14 | 10 | 280.51 | 10 | 2285.07 | 10 | 280.53 |
| GraphColouring | – | – | – | – | 3 | 510.15 | 3 | 531.83 |
| HanoiTower | – | – | – | – | 8 | 408.31 | 9 | 474.39 |
| HydraulicLeaking | 10 | 1936.64 | 7 | 2830.51 | 10 | 2130.24 | 7 | 2843.08 |
| HydraulicPlanning | 10 | 837.25 | 10 | 164.01 | 10 | 889.13 | 10 | 173.13 |
| KnightTour | 5 | 711.54 | 5 | 15.81 | 9 | 1002.47 | 10 | 1092.94 |
| Labyrinth | 3 | 71.48 | 3 | 71.36 | 5 | 49.66 | 6 | 21.92 |
| MazeGeneration | 8 | 629.47 | 8 | 472.77 | 10 | 37.24 | 10 | 7.16 |
| MCSQuerying | 10 | 31.08 | 10 | 160.17 | 10 | 138.18 | 6 | 56.58 |
| Numberlink | 4 | 5.50 | 4 | 10.08 | 8 | 130.06 | 8 | 80.31 |
| PackingProblem | 8 | 1910.86 | – | – | 9 | 2215.05 | – | – |
| PartnerUnitsPolynomial | 1 | 443.06 | – | – | 8 | 264.21 | – | – |
| Reachability | 10 | 58.89 | 4 | 69.51 | 5 | 302.53 | 5 | 1201.39 |
| SokobanDecision | 6 | 182.42 | 6 | 280.80 | 10 | 1306.88 | 9 | 926.54 |
| Solitaire | 4 | 85.50 | – | – | 5 | 216.67 | 5 | 109.06 |
| StableMarriage | – | – | – | – | – | – | – | – |
| WeightAssignmentTree | 10 | 549.09 | – | – | 5 | 113.23 | 1 | 12.07 |

**Table 2.** Results of a pool of solvers using different grounders on the instances evaluated at the 3rd ASP Competition. Notice that "GrammarBasedIE" and "MCSQuerying" are shorthands for the problems named "GrammarBasedInformationExtraction" and "MultiContextSystemQuerying", respectively.

and bottom – organized as follows. The first column reports the problem name, and it is followed by two group of columns. Each group is labeled with the considered solver name, and it is composed of two sub-groups, denoting the grounder (groups "DLV-G" and "GRINGO"). Finally, each sub-group is composed of two columns, in which we report the total amount of solved instances and the total CPU time (in seconds) spent to solve them (columns "#" and "Time", respectively).

Looking at Table 2, concerning the results of CLASP, we report that it was able to solve 141 (out of 180) instances using the DLV-G grounder, while it tops to 107 using GRINGO. In particular, looking at the table, we can see that CLASP mainly benefits from the usage of DLV-G – in terms of total amount of solved instances – in DisjunctiveScheduling, PackingProblem, PartnerUnitsPolynomial, and WeightAssignmentTree problems. Looking at the performance of CMODELS, we can see a very similar picture; notice that DLV-G +CMODELS solves 132 instances, while GRINGO +CMODELS stops at 99. While we can find the same picture looking at IDP performance (it solves 135 formulas if coupled with DLV-G, while 104 if coupled with GRINGO), the picture changes in a noticeable way if we look at the performance of DLV, because – as expected – it performs better using its native grounder (in terms of total amount of solved instances) instead of GRINGO – it solves 100 instances instead of 68. Looking at results in which a solver solves the same number of instances with both grounders, we report that in most cases the usage of GRINGO leads to lower CPU times.

To investigate this phenomenon and possibly getting advantage on this picture, we computed some problem characteristics called *features*. This is the first proposal of *non-ground* ASP features in ASP solving: our choice is to compute "simple" but meaningful features, that are cheap-to-compute, and that can help to discriminate among problems and/or classes, in order to employ the "best" grounder on each instance considered.

Most of the features we extract are related to peculiarities of ASP that can lead to select the most appropriate grounders, e.g. if the program contains a query we want to choose DLV-G given it implements specialized techniques to deal with ASP programs with queries [**?**]. Such features are: fraction of non-ground rules, either normal or disjunctive, presence of queries, ground and partially grounded queries; maximum Strongly Connected Components (SCC) size, number of Head-Cycle Free (HCF) and non-HCF components, degree of support for non-HCF components; features indicating if the program is recursive, tight and stratified; and number of builtins. This set of ASP peculiar features is complemented with features that take into account other characteristics such as problem size, balancing measures and proximity to horn, and are the following: number of predicates, maximum body size, ratio of positive and negative literals in each body of non-ground rules, and its reciprocal, fraction of unary, binary and ternary non-ground rules, and fraction of horn rules.

In order to highlight the differences between DLV-G and GRINGO, we extract the features described above on the instances *submitted* at the 3rd ASP Competition, i.e., a pool of more than one thousand instances related to the problems listed in Table 1, from which we discarded the instances involved in the experiment of Table 2. In Figure 1 we report the boxplots related to the distribution of two features. The differences between DLV-G and GRINGO herewith reported motivates the work presented in the next section.

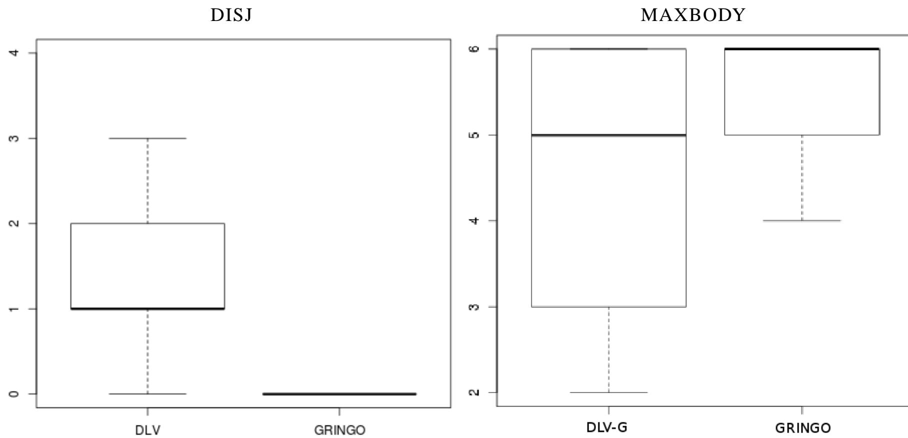


**Fig. 1.** Distributions of the features: number of disjunctive rules (DISJ) (left) and maximum body size (MAXBODY) (right) considering problems *submitted* to the 3rd ASP Competition for which DLV-G allows a better performance with respect to GRINGO (distribution on the left of each plot) and vice-versa (distribution of the right of each plot). For each distribution, we show a box-and-whiskers diagram representing the median (bold line), the first and third quartile (bottom and top edges of the box), the minimum and maximum (whiskers at the top and the bottom) of a distribution.

## 5 Implementation and Experiments

In this section we show the implementation of our automated grounder selector, representing a first attempt towards the exploitation of automated selection techniques to predict both grounder and solver, given an ASP non-ground instance. Our current implementation is composed of two main elements, namely a feature extractor able to analyze non-ground ASP programs, and a decision making module. In particular, the latter has been implemented as an if-then-else decision list, computed with the support of the PART decision list generator [**?**], a classifier that returns a human readable model based on if-then-else rules.

The resulting model highlights some specific cases in which there is a clear difference in performance between two grounders. DLV-G is always chosen when one has to deal with queries. GRINGO is usually preferable for instantiating non-disjunctive and recursive encodings with many components, and is preferable, in particular, when most of the rules of the encoding feature a short body. DLV-G is usually the right choice also when the encoding contains rules having large bodies (say, bodies with 4 or more literals) and the program has a simple structure (few components). The reasons behind this result can be found both in the techniques implemented in the two grounders and in some more specific implementation choices. For instance queries are processed far better by DLV-G, which exploits specific techniques like magic sets [**?**],

| Solver | Grounder | *P* | | *NP* | | **Total** | |
|---|---|---|---|---|---|---|---|
| | | # | Time | # | Time | # | Time |
| CLASP | DLV-G | 48 | 128.26 | 93 | 62.65 | 141 | 84.99 |
| | GRINGO | 35 | 97.21 | 72 | 32.69 | 107 | 53.80 |
| | SELECTOR | 48 | 70.94 | 95 | 59.64 | 143 | 63.43 |
| CMODELS | DLV-G | 46 | 130.47 | 86 | 82.42 | 132 | 99.16 |
| | GRINGO | 32 | 116.29 | 67 | 58.44 | 99 | 77.14 |
| | SELECTOR | 46 | 70.60 | 87 | 80.72 | 133 | 77.22 |
| DLV | DLV-G | 41 | 129.17 | 59 | 71.39 | 100 | 95.08 |
| | GRINGO | 31 | 107.89 | 37 | 28.53 | 68 | 64.71 |
| | SELECTOR | 41 | 71.26 | 59 | 69.50 | 100 | 70.22 |
| IDP | DLV-G | 43 | 136.54 | 92 | 71.13 | 135 | 91.96 |
| | GRINGO | 32 | 140.57 | 72 | 46.97 | 104 | 75.77 |
| | SELECTOR | 43 | 74.16 | 94 | 70.19 | 137 | 71.43 |

**Table 3.** Performance of a pool of ASP solvers combined with DLV-G, GRINGO, and SELECTOR.

while this is a feature that is not well supported in GRINGO.[8] GRINGO is a newer implementation which (we argue) was initially optimized do deal with non-disjunctive encodings, since also CLASP (the solver developed by the same team) does not support disjunction. Finally, we argue that DLV-G performs better with rules having comparatively long bodies because it features both a more sophisticated indexing technique (w.r.t. GRINGO) and effective join ordering heuristics [**?**]; indeed, these techniques are expected to pay off especially in these cases. The grounder selector presented in this paper is available for download at `http://www.mat.unical.it/ricca/downloads/GR-SELECTOR-AIIA.zip`.

Aim of our next experiment is to test the performance of the pool of solvers introduced in Section 4 using the proposed tool as grounder (called SELECTOR in the following). Table 3 shows the results of the experiment described above on the benchmark instances evaluated at the 3rd ASP Competition. The table is structured as follows: "Solver" and "Grounder" report the solver and the grounder name, respectively; for each grounder+solver S, "P", "NP" report the number of instance solved by S ("#") and the average CPU time (in seconds) spent on such instances ("Time") related to the benchmark instances comprised in the classes *P* and *NP*, respectively.

Looking at Table 3, we can see that all the considered solvers benefit from the usage of SELECTOR as grounder. Looking at the performance of CLASP, we can see that SELECTOR+CLASP it is able to solve 2 instances more DLV-G +CLASP, and 36 instances more that GRINGO +CLASP. The increase of performance is noticeable concerning *NP* instances, while if we look at the performance of *P* instance, we report that SELECTOR+CLASP is able to solve the same instances of DLV-G +CLASP, also if in this case

[8] In the competition, as well as in our experiment, only ground queries are used and when calling GRINGO a straight technique is employed to handle propositional queries: $q?$ is replaced by the constraint $:-$ `not` $q$, concluding the $q$ is cautiously true when the resulting program has no answer set.

the average CPU time per solved instance related to SELECTOR+CLASP is 55% of the one related to DLV-G +CLASP.

Looking now at the performance of CMODELS, we can see that the picture is very similar to the one related to CLASP. SELECTOR+CMODELS solves 34 instances more that GRINGO +CMODELS, while the gap with DLV-G +CMODELS stops to 1. Similar considerations can be reported in the case of IDP. Finally, considering the performance of DLV, we can see that the picture slightly changes. In this case, SELECTOR+DLV is never superior to DLV-G +DLV– in terms of total amount of solved instances – but we report better performance in terms of average CPU time per solved instance, i.e., SELECTOR+DLV is about 25% faster than DLV-G +DLV.

## 6 Conclusions

ASP systems are obtained combining a grounder, which eliminates variables, and a solver, that computes the answer sets. It is well known that both components play a central role in the performance of the system. Algorithm selection techniques, up to now, have been applied only on the second component. In this paper we make a first step toward the exploitation of automated selection techniques to the grounding component.

In particular, we implemented a new system able to automatically select the most appropriate grounder for solving the instance at hand out of two state-of-the-art ASP instantiators. An experimental analysis, conducted on benchmarks and solvers from the 3rd ASP Competition, shows that our grounder selector improves the evaluation performance independently from the solver associated.

As far as future work is concerned we are exploring the possibility to implement a selector that is able to predict the best grounder+solver pair among a set of possible combinations.