

# Optimizing Answer Set Computation via Heuristic-Based Decomposition \*

Francesco Calimeri, Simona Perri and Jessica Zangari

*Department of Mathematics and Computer Science, University of Calabria,  
Rende, Italy*

(e-mail: {calimeri,perri,zangari}@mat.unical.it)

*submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003*

---

## Abstract

Answer Set Programming (ASP) is a purely declarative formalism developed in the field of logic programming and nonmonotonic reasoning: computational problems are encoded by logic programs whose answer sets, corresponding to solutions, are computed by an ASP system. Different, semantically equivalent, programs can be defined for the same problem; however, performance of systems evaluating them might significantly vary. We propose an approach for automatically transforming an input logic program into an equivalent one that can be evaluated more efficiently. One can make use of existing tree-decomposition techniques for rewriting selected rules into a set of multiple ones; the idea is to guide and adaptively apply them on the basis of proper new heuristics, to obtain a smart rewriting algorithm to be integrated into an ASP system. The method is rather general: it can be adapted to any system and implement different preference policies. Furthermore, we define a set of new heuristics tailored at optimizing grounding, one of the main phases of the ASP computation; we use them in order to implement the approach into the ASP system *DLV*, in particular into its grounding subsystem  *$\mathcal{S}$ -DLV*, and carry out an extensive experimental activity for assessing the impact of the proposal.

**KEYWORDS:** Answer Set Programming, Artificial Intelligence, ASP in practice

---

## 1 Introduction

Answer Set Programming (ASP) (Brewka et al. 2011; Gelfond and Lifschitz 1991) is a declarative programming paradigm proposed in the area of non-monotonic reasoning and logic programming. With ASP, computational problems are encoded by logic programs whose answer sets, corresponding to solutions, are computed by an ASP system (Lifschitz 1999).

The evaluation of ASP programs is “traditionally” split into two phases: *grounding*, that generates a propositional theory semantically equivalent to the input program, and *solving*, that applies propositional techniques for computing the intended semantics (Alviano et al. 2017; Gebser

---

\* This work is the extended version of a paper originally appeared in the Proceedings of 20th Symposium on Practical Aspects of Declarative Languages (PADL 2018), January 8–9, 2018, Los Angeles, USA. Program chairs were Kevin Hamlen and Nicola Leone. The paper presents new material that integrates and extends what has been reported in the original paper; in particular, it provides the reader with proper preliminaries (omitted in the original paper for space constraints), more detailed discussions on the proposed techniques and richer comparisons with related approaches, along with an extended number of examples. Furthermore, a more thorough experimental activity is presented, discussed in part in the main text and in part in the appendices, that covers also new domains that were unexplored in the original paper.

et al. 2015; Kaufmann et al. 2016; Leone et al. 2006); nevertheless, in the latest years several approaches that deviate from this schema have been proposed (Palù et al. 2009; Dao-Tran et al. 2012; Eiter et al. 2017; Lefèvre et al. 2017).

Typically, the same computational problem can be encoded by means of many different ASP programs which are semantically equivalent; however, real ASP systems may perform very differently when evaluating each one of them. This behavior is due, in part, to specific aspects, that strictly depend on the ASP system employed, and, in part, to general “intrinsic” aspects, depending on the program at hand which could feature some characteristics that can make computation easier or harder. Thus, often, to have satisfying performance, expert knowledge is required in order to select the best encoding. This issue, in a certain sense, conflicts with the declarative nature of ASP that, ideally, should free the users from the burden of the computational aspects. For this reason, ASP systems tend to be endowed with proper pre-processing means aiming at making performance less encoding-dependent; intuitively, this is crucial for fostering and easing the usage of ASP in practice.

A proposal in this direction is *lpopt* (Bichler et al. 2016a), a pre-processing tool for ASP systems that rewrites rules in input programs by means of *tree-decomposition* algorithms. The rationale comes from the fact that, when programs contain rules featuring long bodies, ASP systems performance might benefit from a careful split of such rules into multiple, smaller ones. However, it is worth noting that, while in some cases such decomposition is convenient, in other cases keeping the original rule is preferable; hence, a black-box decomposition, like the one of *lpopt*, makes it difficult to predict whether it will lead to benefits or disadvantages.

Inspired by the idea implemented in *lpopt* of rewriting ASP programs by means of tree-decomposition, we propose here a method that aims at taking full advantage from rewriting, still avoiding performance drawbacks by estimating its effects in advance. It analyzes each input rule before the evaluation, and estimates whether it is convenient to decompose it into an equivalent set of smaller rules, or not; if more than one decomposition is possible, the most promising is selected. The method is general and defined so that all choices are made according to proper criteria and heuristics that can be customized: it can be tailored to different phases of the ASP computation, and it is not tied to a specific system. Furthermore, we define new heuristics and criteria relying on data and statistics dynamically computed during the instantiation with the aim of optimizing the performances of  $\mathcal{J}$ -DLV (Calimeri et al. 2017b), a recently released deductive database system that currently serves also as the grounding subsystem of DLV (Alviano et al. 2017). In addition, we present here an actual implementation into  $\mathcal{J}$ -DLV and perform an extensive experimental activity in order to assess the effects of our technique on ASP program optimization.

The remainder of the paper is structured as follows. In Section 2 we recall ASP basics along with some other preliminary notions; in Section 3 we introduce an abstract heuristic-guided decomposition algorithm for ASP programs in its general form, while in Section 4 we describe how we adapt it in order to foster an actual implementation into the  $\mathcal{J}$ -DLV grounder, along with custom heuristics for guiding the process. Section 5 presents the results of an extensive experimental activity aimed at assessing the impact of the proposed method, and effectiveness of the proposed heuristics on grounding performance; we also shed a light on the impact on solvers. Our conclusions are drawn in Section 6. Some additional experiments, that have been omitted from the main text for the sake of readability, are reported and discussed in appendices.

## 2 Preliminaries

In this section we provide the reader with some preliminaries; in particular, we first briefly introduce Answer Set Programming and then recall how hypergraphs can be used in order to represent ASP rules along with tree-decomposition strategies for rewriting them.

### 2.1 Answer Set Programming

A significant amount of work has been carried out on extending the basic language of ASP, and the community recently agreed on a standard input language for ASP systems: ASP-Core-2 (Calimeri et al. 2013), the official language of the ASP Competition series (Calimeri et al. 2016; Gebser et al. 2016). For the sake of simplicity, we focus next on the basic aspects of the language; for a complete reference to the ASP-Core-2 standard, and further details about advanced ASP features, we refer the reader to (Calimeri et al. 2013) and the vast literature.

A *term* is either a *simple term* or a *functional term*. A *simple term* is either a constant or a variable. If  $t_1 \dots t_n$  are terms and  $f$  is a function symbol of arity  $n$ , then  $f(t_1, \dots, t_n)$  is a *functional term*. If  $t_1, \dots, t_k$  are terms and  $p$  is a *predicate symbol* of arity  $k$ , then  $p(t_1, \dots, t_k)$  is an *atom*. A *literal*  $l$  is of the form  $a$  or *not*  $a$ , where  $a$  is an atom; in the former case  $l$  is *positive*, otherwise *negative*. A *rule*  $r$  is of the form  $\alpha_1 \mid \dots \mid \alpha_k :- \beta_1, \dots, \beta_n, \text{not } \beta_{n+1}, \dots, \text{not } \beta_m$ , where  $m \geq 0$ ,  $k \geq 0$ ;  $\alpha_1, \dots, \alpha_k$  and  $\beta_1, \dots, \beta_m$  are atoms. We define  $H(r) = \{\alpha_1, \dots, \alpha_k\}$  (the *head* of  $r$ ) and  $B(r) = B^+(r) \cup B^-(r)$  (the *body* of  $r$ ), where  $B^+(r) = \{\beta_1, \dots, \beta_n\}$  (the *positive body*) and  $B^-(r) = \{\text{not } \beta_{n+1}, \dots, \text{not } \beta_m\}$  (the *negative body*). If  $H(r) = \emptyset$  then  $r$  is a (*strong*) *constraint*; if  $B(r) = \emptyset$  and  $|H(r)| = 1$  then  $r$  is a *fact*. A rule  $r$  is *safe* if each variable of  $r$  has an occurrence in  $B^+(r)$ <sup>1</sup>. For a rule  $r$ , we denote as *headvar*( $r$ ), *bodyvar*( $r$ ) and *var*( $r$ ) the set of variables occurring in  $H(r)$ ,  $B(r)$  and  $r$ , respectively. An ASP program is a finite set  $P$  of safe rules. A program (a rule, a literal) is *ground* if it contains no variables. A predicate is defined by a rule  $r$  if it occurs in  $H(r)$ . A predicate defined only by facts is an *EDB* predicate, the remaining are *IDB* predicates. The set of all facts in  $P$  is denoted by *Facts*( $P$ ); the set of instances of all *EDB* predicates in  $P$  is denoted by *EDB*( $P$ ).

Given a program  $P$ , the *Herbrand universe* of  $P$ , denoted by  $U_P$ , consists of all ground terms that can be built combining constants and function symbols appearing in  $P$ . The *Herbrand base* of  $P$ , denoted by  $B_P$ , is the set of all ground atoms obtainable from the atoms of  $P$  by replacing variables with elements from  $U_P$ . A *substitution* for a rule  $r \in P$  is a mapping from the set of variables of  $r$  to the set  $U_P$  of ground terms. A *ground instance* of a rule  $r$  is obtained applying a substitution to  $r$ . The *full instantiation* *Ground*( $P$ ) of  $P$  is defined as the set of all ground instances of its rules over  $U_P$ . An *interpretation*  $I$  for  $P$  is a subset of  $B_P$ . A positive literal  $a$  (resp., a negative literal *not*  $a$ ) is true w.r.t.  $I$  if  $a \in I$  (resp.,  $a \notin I$ ); it is false otherwise. Given a ground rule  $r$ , we say that  $r$  is satisfied w.r.t.  $I$  if some atom appearing in  $H(r)$  is true w.r.t.  $I$  or some literal appearing in  $B(r)$  is false w.r.t.  $I$ . Given a program  $P$ , we say that  $I$  is a *model* of  $P$ , iff all rules in *Ground*( $P$ ) are satisfied w.r.t.  $I$ . A model  $M$  is *minimal* if there is no model  $N$  for  $P$  such that  $N \subset M$ . The *Gelfond-Lifschitz reduct* (Gelfond and Lifschitz 1991) of  $P$ , w.r.t. an interpretation  $I$ , is the positive ground program  $P^I$  obtained from *Ground*( $P$ ) by: (i) deleting all rules having a negative literal false w.r.t.  $I$ ; (ii) deleting all negative literals from the remaining

<sup>1</sup> We remark that this definition of safety is specific for the syntax considered herein. For a complete definition we refer the reader to (Calimeri et al. 2013).

rules.  $I \subseteq B_P$  is an *answer set* for a program  $P$  iff  $I$  is a minimal model for  $P^I$ . The set of all answer sets for  $P$  is denoted by  $AS(P)$ .

## 2.2 ASP Computation

The high expressiveness of ASP comes at the price of a high computational cost in the worst case (Eiter et al. 1997; Leone et al. 2006), which makes the implementation of efficient ASP systems a difficult task. Thanks to the effort by a scientific community that grew over time, there are nowadays a number of systems that support ASP and its variants (Simons et al. 2002; Ward and Schlipf 2004; Janhunen et al. 2006; Giunchiglia et al. 2006; Gebser et al. 2012; Alviano et al. 2015; Leone et al. 2006; Gebser et al. 2014; Palù et al. 2009; Lefèvre et al. 2017; Dao-Tran et al. 2012; Weinzierl 2017).

The well-established, mainstream approach for the evaluation of ASP programs (Kaufmann et al. 2016) relies on two phases, usually referred to as *instantiation* or *grounding*, and *solving* or *answer sets search*, respectively. Given a (non-ground) ASP program  $P$ , grounding consists of producing a propositional theory  $G_P$  semantically equivalent to  $P$ , i.e. such that  $G_P$  does not contain any variable but has the same answer sets as  $P$ . Given that, in the worst case, the solving stage may take up to exponential time in the size of  $G_P$  (Ben-Eliyahu and Dechter 1994; Ben-Eliyahu-Zohary and Palopoli 1997), modern ASP systems employ intelligent grounding procedures so that  $G_P$  is significantly smaller than the full instantiation  $Ground(P)$ . Once the program  $G_P$  has been computed, solving takes place, taking as input  $G_P$  and computing its answer sets by means of propositional algorithms. The majority of current ASP implementations follows this two-phase computation, either by explicitly relying on stand-alone grounders (Syrjänen 2001; Faber et al. 2012; Gebser et al. 2011) and solvers (Simons et al. 2002; Ward and Schlipf 2004; Janhunen et al. 2006; Giunchiglia et al. 2006; Gebser et al. 2012; Alviano et al. 2015), or integrating the modules into monolithic systems (Gebser et al. 2014; Leone et al. 2006; Alviano et al. 2017). Notably, given that both phases are, in general, computationally expensive (Eiter et al. 1997; Dantsin et al. 2001), efficient ASP implementations depend on proper optimization of both.

Alternative solutions (Palù et al. 2009; Lefèvre et al. 2017; Dao-Tran et al. 2012; Weinzierl 2017) adopt a *lazy grounding* technique, in which grounding and solving steps are interleaved, and rules are grounded on-demand during solving. These systems try to overcome the so called *grounding bottleneck*, that occurs on problems for which the instantiation is inherently so huge that its actual materialization is not suitable in practice. For this reason, this approach looks promising; however, current implementations do not match, in the general case, performance of the more “traditional” systems, that proved instead to be reliable and well-performing in a wider range of scenarios.

Notably, the herein presented technique, introduced in Section 3, is general enough to be adopted with both approaches, by defining suitable heuristics and properly customizing its integration.

## 2.3 Tree-Decompositions for Rewriting ASP Rules

Hypergraphs are useful for describing the structure of many computational problems; furthermore, it is possible to decompose them into different parts, so that the solution(s) of problems can be obtained by a polynomial divide-and-conquer algorithm that properly exploits this di-

vision (Gottlob et al. 2001; Gottlob et al. 2005). Such ideas can guide a rewriting of an ASP program: indeed, a logic rule can be represented as a *hypergraph* (Morak and Woltran 2012), and hence properly decomposed.

Discussing in detail how tree decompositions can be computed and rewritings induced is out of the scope of this paper; indeed, our main goal is to find a way for correctly identifying in advance in which cases their application pays off in terms of efficiency, while dealing with ASP rules. However, in order to ease the reading, in the following, we briefly recall an intuitive description of some crucial notions for the ASP context; for further details we refer the reader to (Bichler et al. 2016a) and the existing literature.

A (undirected) hypergraph is a generalization of a (undirected) graph in which an edge can join two or more vertices. An ASP rule  $r$  can be represented as a hypergraph  $HG(r)$  by adding a hyperedge for each literal  $l \in B(r) \cup H(r)$  containing the variables appearing in  $l$ . A tree decomposition of a hypergraph  $HG(r)$  (see (Robertson and Seymour 1986; Gottlob et al. 2016)) is a tuple  $(TD(r), \chi)$ , where  $TD(r) = (V(TD(r)), E(TD(r)))$  is a tree and  $\chi : V(TD(r)) \rightarrow 2^{V(HG(r))}$  is a function associating a set of vertices  $\chi(t) \subseteq V(HG(r))$  to each vertex  $t$  of the decomposition tree  $TD(r)$ , such that for each  $e \in E(HG(r))$  there is a node  $t \in V(TD(r))$  such that  $e \subseteq \chi(t)$ , and for each  $v \in V(HG(r))$  the set  $\{t \in V(TD(r)) \mid v \in \chi(t)\}$  is connected in  $TD(r)$ . Intuitively, a tree decomposition  $TD(r)$  of  $HG(r)$  is a tree such that each vertex is associated to a *bag*, i.e., a set of nodes of  $HG(r)$ , and such that each hyperedge of  $HG(r)$  is covered by some bag, and for each node of  $HG(r)$  all vertices of  $TD(r)$  whose bag contains it induce a connected subtree of  $TD(r)$ .

A tree decomposition  $TD(r)$  can be used in order to produce a set of rules that rewrites  $r$ ; such set is called *rule decomposition*, and denoted by  $RD(r)$ . In particular,  $RD(r)$  contains a (newly generated) rule for each vertex  $v$  of  $TD(r)$ , on the basis of the included variables. Roughly, each literal  $l$  in the body of  $r$ , such that the set of variables in  $l$  is contained in  $v$ , is added to the body of the rule generated for  $v$ . Eventually, some optional rules may be added to  $RD(r)$  in order to guarantee safety. Note that, since different choices for handling safety can be performed, the way in which a tree decomposition is converted into a rule decomposition might be not unique. Moreover, interestingly, in general, more than one decomposition is possible for each rule.

The following running example, which we will refer to throughout the paper, illustrates this mechanism.

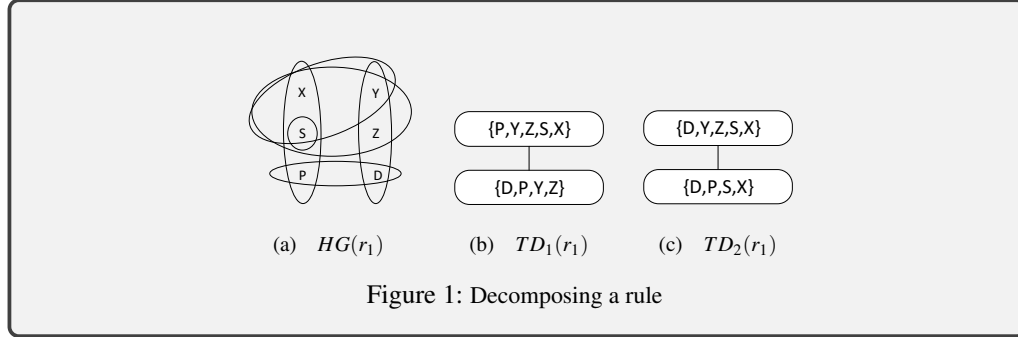
#### Example 1

Let us consider the rule:

$$r_1 : p(X, Y, Z, S) : \neg s(S), a(X, Y, S - 1), c(D, Y, Z), f(X, P, S - 1), P \geq D.$$

from the encoding of the problem *Nomystery* from the 6th ASP Competition (see Section 5), where, for the sake of readability, predicates and variables have been renamed. Figure 1 depicts the conversion of  $r_1$  into the hypergraph  $HG(r_1)$ , along with two possible decompositions:  $TD_1(r_1)$  and  $TD_2(r_1)$ , that induce two different rewritings. According to  $TD_1(r_1)$ ,  $r_1$  can be rewritten into the set of rules  $RD_1(r_1)$ :

$$\begin{aligned} r_2 &: p(X, Y, Z, S) : \neg s(S), a(X, Y, S - 1), f(X, P, S - 1), \text{fresh\_pred\_1}(P, Y, Z). \\ r_3 &: \text{fresh\_pred\_1}(P, Y, Z) : \neg c(D, Y, Z), P \geq D, \text{fresh\_pred\_2}(P). \\ r_4 &: \text{fresh\_pred\_2}(P) : \neg s(S), f(\_, P, S - 1). \end{aligned}$$



In particular, the rule  $r_2$  features the same head of  $r_1$  and as body the literals needed in order to cover the node of  $TD_1(r_1)$  containing the variables  $\{P, Y, Z, S, X\}$ ;  $r_3$  features as head the fresh predicate *fresh\_pred\_1* that links it to  $r_2$  and collects in its body a set of literals covering the variables  $\{D, P, Y, Z\}$  appearing in the other node of  $TD_1(r_1)$ ; eventually,  $r_4$  is needed to ensure safety of  $r_3$ : the atom *fresh\_pred\_2*( $P$ ) is added in the body of  $r_3$  and to the head of  $r_4$ , whose body features a set of literals coming from  $r_1$  and covers  $P$  (note that in this case the set is unique). Note that, a different rewriting could be obtained by differently handling safety of  $r_3$ ; for instance, one could avoid to introduce  $r_4$  and, instead, add the literals  $s(S)$ , and  $f(\_, P, S - 1)$  to the body of  $r_3$ .

Similarly, according to  $TD_2(r_1)$ ,  $r_1$  can be rewritten into  $RD_2(r_1)$  as follows:

$$\begin{aligned}
 r_5 &: p(X, Y, Z, S) : -a(X, Y, S - 1), c(D, Y, Z), \text{fresh\_pred\_1}(D, S, X). \\
 r_6 &: \text{fresh\_pred\_1}(D, S, X) : -s(S), f(X, P, S - 1), P \geq D, \text{fresh\_pred\_2}(D). \\
 r_7 &: \text{fresh\_pred\_2}(D) : -c(D, \_, \_).
 \end{aligned}$$

### 3 A Heuristic-guided Decomposition Algorithm

In the previous section we recalled how tree decomposition of hypergraphs can be used in order to guide rewritings of ASP rules. Interestingly, the *lpopt* (Bichler et al. 2016a) preprocessor is a proposal in this direction, that rewrites an ASP program before it is fed to an ASP system.

As previously noted, for each rule, several different rule decompositions might exist. However, when fed to a real ASP system, different yet equivalent rewritings require, in general, significantly different evaluation times. Thus, proper means for reasonably and effectively choose the “best” rewriting are crucial; furthermore, it might be the case that, whatever the choice, sticking to the original, unrewritten rule, is still preferable. Hence, a black-box approach, such as the one of *lpopt*, makes it difficult to effectively take advantage from the decomposition rewritings; this is clearly noticeable by looking at experiments, as discussed in Section 5.

In this section we introduce a smart decomposition algorithm that aims at addressing the above issues; it is designed to be integrated into an ASP system, and uses information available during the computation to predict, according to proper criteria, whether decomposing will pay off or not; moreover, it chooses the most promising decomposition, among the several possible ones. In the following, we first describe the method in its general form, that can be easily adapted to different real systems; a complete actual implementation, specialized for the *DLV* system, is presented later on.

```

function SMARTDECOMPOSITION( $r$  : Rule) : RuleDecomposition
  var  $e_r$  : number,  $RDS$  : SetOfRuleDecompositions,  $e_{RD}$  : Number,
   $RD$  : RuleDecomposition
   $e_r \leftarrow$  ESTIMATE( $r$ )
   $RDS \leftarrow$  GENERATERULEDECOMPOSITIONS( $r$ )
  if  $RDS \neq \emptyset$  then                                     /*  $r$  is decomposable */
     $RD \leftarrow$  CHOOSEBESTDECOMPOSITION( $RDS, e_r$ )
     $e_{RD} \leftarrow$  ESTIMATEDecomposition( $RD$ )
    if DECOMPOSITIONISPREFERABLE( $e_r, e_{RD}$ ) then
      return  $RD$ 
    end if
  end if
  return  $\emptyset$ 
end function

function GENERATERULEDECOMPOSITIONS( $r$  : Rule) : SetOfRuleDecompositions
  var  $HG$  : Hypergraph,  $RDS$  : SetOfRuleDecompositions,
   $RD$  : RuleDecomposition,  $TD$  : TreeDecomposition
   $TDS$  : SetOfTreeDecompositions
   $HG \leftarrow$  TOHYPERGRAPH( $r$ )
   $TDS \leftarrow$  GENERATETREEDecompositions( $HG$ )
  for each  $TD \in TDS$  do
     $RD \leftarrow$  TORULES( $TD, r$ )
     $RDS = RDS \cup \{RD\}$ 
  end for
  return  $RDS$ 
end function

```

Figure 2: The algorithm SMARTDECOMPOSITION and the GENERATERULEDECOMPOSITIONS function.

The abstract algorithm SMARTDECOMPOSITION is shown in Figure 2; we indicate as *tree decomposition* an actual tree decomposition of a hypergraph, while with *rule decomposition* we denote the conversion of a tree decomposition into a set of ASP rules. Given a (non-ground) input rule  $r$ , the algorithm first heuristically computes, by means of the ESTIMATE function, a value  $e_r$  that estimates how much the presence of  $r$  in the program impacts on the whole computation; then, the function GENERATERULEDECOMPOSITIONS computes a set of possible rule decompositions  $RDS$ , from which CHOOSEBESTDECOMPOSITION selects the best  $RD \in RDS$ ; hence, function ESTIMATEDecomposition computes the value  $e_{RD}$  that estimates the impact of having  $RD$  in place of  $r$  in the input program. Eventually, function DECOMPOSITIONISPREFERABLE is in charge of comparing  $e_r$  and  $e_{RD}$  and deciding if decomposing is convenient. We remark that functions ESTIMATE, CHOOSEBESTDECOMPOSITION, ESTIMATEDecomposition and DECOMPOSITIONISPREFERABLE are left unimplemented, as they are completely customizable; they must be implemented by defining proper criteria that take into account features and information within the specific evaluation procedure, and the actual ASP system the algorithm is being integrated into.

Figure 2 reports also the implementation of function GENERATERULEDECOMPOSITIONS. Here, TOHYPERGRAPH converts a input rule  $r$  into a hypergraph  $HG$ , which is iteratively analysed in order to produce possible tree decompositions, by means of the function GENERATE-

TREEDecompositions. Also these stages can be customized in an actual implementation, according to different criteria and the features of the system at hand; for space reasons, we refrain from going into details that are not relevant for the description of the approach. The function TORULES, given a tree decomposition  $TD$  and a rule  $r$ , converts  $TD$  into a rule decomposition  $RD$  for  $r$ . In particular, for each node in  $TD$ , it adds a new logic rule to  $RD$ , possibly along with some additional auxiliary rules needed for ensuring safety. The process is, again, customizable, and should be defined according to the function TOHYPERGRAPH.

The general definition of the algorithm provided so far is independent from any actual implementation, and its behaviour can significantly change depending on the customization choices, as discussed above. However, in order to give an intuition on how it works, we make use of our running example for illustrating a plausible execution.

#### Example 2

Given rule  $r_1$  of Example 1, let us imagine that function GENERATERULEDECOMPOSITIONS computes the tree decompositions  $TD_1(r_1)$  and  $TD_2(r_1)$  and then, by means of TORULES, the set of rule decompositions consisting of  $RD_1(r_1)$  and  $RD_2(r_1)$  is generated. Note that  $r_4$  and  $r_7$  are added for ensuring safety of rules  $r_3$  and  $r_6$ , respectively. Next step consists of the choice between  $RD_1(r_1)$  and  $RD_2(r_1)$  for the best promising decomposition, according to the actual criteria of choice. Supposing that it is  $RD_1(r_1)$ , DECOMPOSITIONISPREFERABLE compares the estimated impacts  $e_{r_1}$  and  $e_{RD_1(r_1)}$ , in order to decide if keeping  $r$  or substituting it with  $RD_1(r_1)$ .

### 4 Integrating the SMARTDECOMPOSITION Algorithm into a Real System: the DLV Case

In this section we illustrate how the general SMARTDECOMPOSITION algorithm of Section 3 can be customized in order to be integrated into an actual ASP implementation. Interestingly, such customization can be tailored with different purposes, for both the two-phase-based and the lazy-grounding-based systems, for optimizing solving or instantiation performance, according to different criteria (times, size, structure, etc.). In this work, we focus on the widespread DLV system (Alviano et al. 2017; Leone et al. 2006), which complies to the two-phase strategy, with the explicit aim of optimizing performance of its grounding subsystem  $\mathcal{J}$ -DLV (Calimeri et al. 2017b). A detailed description of the  $\mathcal{J}$ -DLV computation is out of the scope of this work (the interested reader is referred to (Calimeri et al. 2017b)); however, for the sake of readability, we briefly recall the basics of its machinery.

Given an ASP program  $P$ :

1.  $P$  is parsed, and the extensional database (EDB) is built.
2. Each rule in  $P$  is analyzed, and possibly rewritten according to different strategies for optimization purposes; the result constitutes the intensional database (IDB).
3. Dependencies among IDB rules and predicates are examined; such dependencies induce the splitting of  $P$  into modules, and a suitable processing ordering is computed so that an incremental evaluation is possible according to the definitions in (Faber et al. 2012).
4. The program is grounded one module at a time by means of a proper adaptation of a semi-naïve schema (Faber et al. 2012; Ullman 1988) that evaluates each rule in a module according to a rule instantiation procedure that in turn produces its ground instances. Rules within a module can be *recursive* or not. While for the former ones the procedure might be iteratively invoked, for the not recursive rules a single call of the rule instantiation procedure is enough to produce all their ground instances.



5. The collection of the ground rules generated from all *IDB* rules compose, along with *EDB*(*P*), the resulting ground program  $G_P$ .

The *core* of the  $\mathcal{J}$ -DLV computation is the rule instantiation process mentioned in the step 4 of the sketch above, which constitutes one of the more computationally heavy tasks. Basically, when grounding a rule  $r$  of  $P$ , instead of replacing  $bodyvar(r)$  with every possible constant appearing in  $P$ , the rule instantiation iteratively substitutes the variables in each body literal with constants appearing in the corresponding predicate extension. A predicate extension of a predicate  $p$  is the set of all ground atoms having  $p$  as predicate. More in detail, given a rule  $r$  and the set of extensions of its body predicates, the rule instantiation produces ground instances of  $r$  by iterating on positive body literals<sup>2</sup> and looking for all possible valid substitutions. Intuitively, this phase resembles the evaluation of relational joins on the positive body literals, where predicate extensions can be seen as tables whose tuples consist of the ground instances. Once a valid substitution is found for all variables in  $bodyvar(r)$ , it is applied to  $headvar(r)$  in order to obtain a totally ground rule, i.e. a ground instance of  $r$ , say  $r'$ . This possibly leads to the generation of new ground atoms occurring in the head of  $r'$ ; such new ground atoms are added to the corresponding predicate extensions. It is worth noting that, the set of all predicate extensions is built dynamically starting from ground atoms appearing in *Facts*( $P$ ) and then, adding each new ground atom coming from heads of produced ground rules; the chosen evaluation order plays a key role in this respect as it ensures that when evaluating a rule  $r$  the extensions of all body predicates needed for instantiating  $r$  have been fully generated.

Besides the basic schema herein sketched,  $\mathcal{J}$ -DLV employs smart optimizations techniques, geared towards the efficient production of a ground program that is considerably smaller, still preserving the semantics. Roughly, when a rule is going to be instantiated,  $\mathcal{J}$ -DLV firstly performs a pre-processing that might lead some adjustments over the rule to different extents, and after that the actual rule instantiation takes place, a post-processing refines the output. Some optimizations, such as, for instance, join-ordering strategies, operate in the pre-processing phase; some explicitly take place during the actual instantiation process, such as non-chronological backtracking; some operate across the two phases, such as indexing techniques for a quick instances retrieval; others take place in the post-processing step, such as the simplification that removes ground rules and literals in the bodies that do not contribute to the semantics.

The SMARTDECOMPOSITION algorithm implementation herein described works in the pre-processing phase.

We provide next some details on how we defined the functions that have been left unimplemented in the general description of Section 3 (ESTIMATE, CHOOSEBESTDECOMPOSITION, ESTIMATEDecomposition and DECOMPOSITIONISPREFERABLE), along with the proposed heuristics, and discuss further implementation issues.

#### 4.1 The ESTIMATE Function

The function ESTIMATE (Figure 3) heuristically measures the cost of instantiating a rule  $r$  before it is actually grounded. To this aim, we propose a heuristics inspired by the ones introduced in the database field (Ullman 1988) and adopted in (Leone et al. 2001) to estimate the size of a join

<sup>2</sup> Because of the safety condition, in order to generate a completely ground instance of  $r$ , it is enough to have a substitution for the variables occurring in the positive literals.

operation. In particular, it relies on statistics over body predicates, such as size of extensions and argument selectivities; we readapted it in order to estimate the cost of grounding a rule as the total number of operations needed in order to perform the task, rather than estimate the size of the join of its body literals. Let  $a = p(t_1, \dots, t_n)$  be an atom; we denote with  $\text{var}(a)$  the set of variables occurring in  $a$ , while  $T(a)$  represents the number of different tuples for  $a$  in the ground extension of  $p$ . Moreover, for each variable  $X \in \text{var}(a)$ , we denote by  $V(X, a)$  the selectivity of  $X$  in  $a$ , i.e., the number of distinct values in the field corresponding to  $X$  over the ground extension of  $p$ . Given a rule  $r$ , let  $\langle a_1, \dots, a_m \rangle$  be the ordered list of atoms appearing in  $B(r)$ , for  $m > 1$ . Initially, the cost of grounding  $r$ , denoted by  $e_r$ , is set to  $T(a_1)$ , then the following formula is iteratively applied up to the last atom in the body in order to obtain the total estimation cost for  $r$ . More in detail, let us suppose that we estimated the cost of joining the atoms  $\langle a_1, \dots, a_j \rangle$  for  $j \in \{1, \dots, m\}$ , and consequently we want to estimate the cost of joining the next atom  $a_{j+1}$ ; if we denote by  $A_j$  the relation obtained by joining all  $j$  atoms in  $\langle a_1, \dots, a_j \rangle$ , then:

$$e_{A_j \bowtie a_{j+1}} = \frac{T(a_{j+1})}{\prod_{X \in \text{idx}(\text{var}(A_j) \cap \text{var}(a_{j+1}))} V(X, a_{j+1})} \cdot \prod_{X \in (\text{var}(A_j) \cap \text{var}(a_{j+1}))} \frac{V(X, A_j)}{\text{dom}(X)} \quad (1)$$

where  $\text{dom}(X)$  is the maximum selectivity of  $X$  computed among the atoms in  $B(r)$  containing  $X$  as variable, and  $\text{idx}(\text{var}(A_j) \cap \text{var}(a_{j+1}))$  is the set of the indexing arguments of  $a_{j+1}$ . We note that, at each step, once the atom  $a_{j+1}$  has been considered,  $V(X, A_{j+1})$ , representing the selectivity of  $X$  in the virtual relation obtained at step  $j + 1$ , has to be estimated in order to be used at next steps:

$$\begin{aligned} V(X, A_{j+1}) &= V(X, A_j) \cdot \frac{V(X, a_{j+1})}{\text{dom}(X)} && \text{if } X \in \text{var}(A_j) \\ V(X, A_{j+1}) &= V(X, a_{j+1}) && \text{otherwise} \end{aligned} \quad (2)$$

Intuitively, the formula tries to determine the cost of grounding  $r$ , by estimating the total number of operations to be performed. In particular, the first factor is intended to estimate how many instances for  $a_{j+1}$  have to be considered, while the second factor represents the reduction in the search space implied by  $a_{j+1}$ . To obtain a realistic estimate, the presence of indexing techniques, used in  $\mathcal{J}$ -DLV to reduce the number of such operations (Calimeri et al. 2017b), has been taken into account.

### Example 3

Let us consider the rule:

$$r_1 : p(X, Y, Z, S) : \neg s(S), a(X, Y, S - 1), c(D, Y, Z), f(X, P, S - 1), P \geq D.$$

of Example 1, and let us assume that we are dealing with an instance that contains the facts<sup>3</sup>:

$$s(1..5). \quad a(1..5, 1..5, 1..5). \quad c(1..5, 1..5, 1..5). \quad f(1..5, 1..5, 1..5).$$

The ESTIMATE function first estimates, by means of Formula (1), the cost of computing the joins  $A_i$ . In this case, denoting by  $a_1 = s(S)$ ,  $a_2 = a(X, Y, S - 1)$ ,  $a_3 = c(D, Y, Z)$  and so on, we have that  $A_1 = s(S)$ ,  $A_2 = A_1 \bowtie a(X, Y, S - 1)$  and it is estimated as:

<sup>3</sup> According to ASP-Core-2 syntax, the term  $(1..k)$  stands for all values from 1 to  $k$ .

```

function ESTIMATE( $r$  : Rule) : Number
    /* Estimate the cost of grounding a rule according to Formula (1) */
end function
function ESTIMATEDecomposition( $RD$  : SetOfRules) : Number
    var  $e_{RD}$  : number
    PREPROCESS( $RD$ )
     $e_{RD} \leftarrow 0$ 
    for each  $r' \in RD$  do
         $e_{RD} = e_{RD} + \text{ESTIMATE}(r')$ 
    end for
    return  $e_{RD}$ 
end function

```

Figure 3: ESTIMATE and ESTIMATEDecomposition as implemented in  $\mathcal{J}$ -DLV

$$e_{A_1 \bowtie A_2} = \frac{T(a_2)}{V(S, a_2)} \cdot \frac{V(S, A_1)}{\text{dom}(S)} = \frac{125}{5} \cdot \frac{5}{5} = 25 = e_{A_2}$$

Then, the formula is used again in order to estimate the cost of the join  $A_3$  between  $A_2$  and  $a_3$ , and so on up to the last join  $A_4$ . At each step, size and variable selectivities for each  $a_i$  are known, while such data for the intermediate relations  $A_i$  are estimated. The size of  $A_2$  is estimated as  $e_{A_2}$ , and selectivities of all variables appearing in  $A_2$  (i.e.,  $X, Y$ , and  $S$ ) are estimated, according to Formula (2), as:

- $V(X, A_2) = 5$  (indeed,  $X \notin \text{var}(A_1)$ )
- $V(Y, A_2) = 5$  (indeed,  $Y \notin \text{var}(A_1)$ )
- $V(S, A_2) = V(S, A_1) \cdot \frac{V(S, a_2)}{\text{dom}(S)} = 5$  (indeed,  $S \in \text{var}(A_1)$ ).

The process is similarly iterated until the end of the body, from left to right.

#### 4.2 The ESTIMATEDecomposition Function

The ESTIMATEDecomposition function is illustrated in Figure 3: after some pre-processing steps, it computes the cost of a given decomposition as the sum of the cost of each rule in it. Let  $r$  be a rule and  $RD = \{r_1, \dots, r_n\}$  be a rule decomposition for  $r$ . In order to estimate the cost of grounding  $RD$ , one must estimate the cost of grounding all rules in  $RD$ . For each  $r_i \in RD$  the estimate is performed by means of Formula (1). Nevertheless, it is worth noting that each  $r_i$ , in addition to predicates originally appearing in  $r$ , denoted as *known predicates*, may contain some *fresh predicates*, generated during the decomposition. Concerning known predicates, thanks to the rule instantiation ordering followed by  $\mathcal{J}$ -DLV, as already pointed out in Section 4, extensions size and selectivity needed for computing the formula are directly available: hence, there is no need for estimations. On the contrary, for fresh predicates, that have been “locally” introduced and do not appear in any of the rules in the original input program, such data is not available, and must be estimated. To this aim, the dependencies among the rules in  $RD$  are analyzed, and an ordering that guarantees a correct instantiation is determined. Such dependencies come out from the definitions in (Faber et al. 2012): rules depending only on known predicates can be grounded first, while rules depending also on new predicates can be grounded only once the rules that define them have been instantiated. Assuming that for the set  $RD$  a correct instan-

tiation order is represented by  $\langle r_1, \dots, r_n \rangle$ , for each  $r'$  in this ordered list, if  $H(r') = p'(t_1, \dots, t_k)$  for  $k \geq 1$ , and if  $p'$  is a fresh predicate, we estimate: (i) the size of the ground extension of  $p'$ , denoted  $T(p')$ , by means of a formula conceived for estimating the size of a join relation, based on criteria that are well-established in the database field and reported in (Leone et al. 2001); (ii) the selectivity of each argument as  $\sqrt[k]{T(p')}$ . Therefore, the procedure **PREPROCESS** invoked in **ESTIMATEDecomposition** (see Figure 3) amounts to preprocess the rules in  $RD$  according to a valid grounding order  $\langle r_1, \dots, r_n \rangle$  to obtain the extension sizes and the argument selectivities for involved fresh predicates, based on the above mentioned formula. Once estimates for fresh predicates are available, the actual estimate of grounding  $RD$  can be performed.

#### Example 4

Let us consider again the rule  $r_1$  of our running Example 1 and its decomposition  $RD_1(r_1)$ :

$$\begin{aligned} r_2 &: p(X, Y, Z, S) : -s(S), a(X, Y, S-1), f(X, P, S-1), \text{fresh\_pred\_1}(P, Y, Z). \\ r_3 &: \text{fresh\_pred\_1}(P, Y, Z) : -c(D, Y, Z), P \geq D, \text{fresh\_pred\_2}(P). \\ r_4 &: \text{fresh\_pred\_2}(P) : -s(S), f(\_, P, S-1). \end{aligned}$$

In order to compute  $e_{RD_1(r_1)}$  we first need to determine a correct evaluation order of the rules in  $RD_1(r_1)$ ; the only valid one is  $\langle r_4, r_3, r_2 \rangle$ . Indeed,  $r_4$  has only known predicates in its body, thus can be evaluated first; the body of  $r_3$  contains, besides to known predicates,  $\text{fresh\_pred\_2}$ , whose estimates will be available just after the evaluation of  $r_4$ ; eventually,  $r_2$  depends also on  $\text{fresh\_pred\_1}$ , whose estimates will be available right after the evaluation of  $r_3$ . Once the estimates for the fresh predicates  $\text{fresh\_pred\_1}$  and  $\text{fresh\_pred\_2}$  are obtained, they are used for computing  $e_{r_4}$ ,  $e_{r_3}$  and  $e_{r_2}$  with Formula (1), and then for obtaining  $e_{RD_1(r_1)} = e_{r_2} + e_{r_3} + e_{r_4}$ .

### 4.3 The CHOOSEBESTDecomposition and DecompositionIsPreferable

#### Functions

The function **CHOOSEBESTDecomposition** estimates the costs of all decompositions of a rule  $r$  by means of **ESTIMATEDecomposition**, and returns the one with the smallest estimated cost; let us denote it by  $RD$ . The function **DecompositionIsPreferable** is then in charge of deciding whether  $RD$  will substitute  $r$ , by relying on  $e_r$  and  $e_{RD}$ , that are the estimated costs associated to  $r$  and  $RD$ , respectively. More in detail, it computes the ratio  $e_r/e_{RD}$ . Intuitively, when the ratio  $e_r/e_{RD} \geq 1$ , decomposing  $r$  is convenient; nevertheless, it is worth remembering that the costs are estimated, and, in particular, as discussed in Section 4.2, the estimate of the cost of a decomposition requires to estimate also the extension of some additional predicates introduced by the rewriting, thus possibly making the estimate less accurate. This leads sometimes to cases in which the decomposition is preferable even when  $e_r/e_{RD} < 1$ . One can try to improve the estimations, in the first place; however, an error margin will always be present. For this reason, in order to reduce the impact of such issue, we decided to experimentally test the effects of the choices under several values of the ratio, and found that decomposition is preferable when  $e_r/e_{RD} \geq 0.5$ , that has also been set as a default threshold in our implementation; of course, the user can play with this at will. We plan to further improve the choice of the threshold by taking advantage from automatic and more advanced methods, such as machine learning guided techniques.

#### Example 5

Let us consider again our running Example 1. At the final step, three possible alternatives are evaluated: (i) leave the rule  $r_1$  as it is (i.e.  $r_1$  is not decomposed), (ii) choose  $RD_1(r_1)$  or (iii)

choose  $RD_2(r_1)$ . Since the nature of the heuristics we implemented into  $\mathcal{J}$ -DLV have the aim of optimizing the grounding process, estimations tightly depend on the instance at hand; hence, choices will possibly vary from instance to instance.

Let us assume that the current instance contains the same facts reported in Example 3. Then, the costs of instantiating  $RD_1(r_1)$  or  $RD_2(r_1)$  are computed according to what discussed in Section 4.2: without reporting all intermediate calculations, we have  $e_{RD_1(r_1)} = 122,945$ , while  $e_{RD_2(r_1)} = 53,075$ . In this case, the best decomposition is obviously  $RD_2(r_1)$ , and it is compared with the option of grounding  $r_1$  as non-decomposed. Again, without reporting all intermediate calculations, we have that the cost  $e_{r_1}$  of grounding  $r_1$  amounts to 390,625; hence, the ratio  $e_{r_1}/e_{RD_2(r_1)}$  is computed as 7.36, and, given that it is greater than 0.5, we prefer to substitute the original rule with the decomposition  $RD_2(r_1)$  (see discussion above).

Interestingly, with a different input instance, things might change. For instance, if the set of input facts for  $f$  changed to  $f(1..20, 1..20, 1..5)$ , the decomposition  $RD_1(r_1)$  would be preferred.

#### 4.4 Fine-Tuning and Further Implementation Issues

In order to implement the SMARTDECOMPOSITION algorithm, one might rely on *lpopt* in order to obtain a rule decomposition for each rule in the program; in particular, this would lead to a straightforward implementation of TOHYPERGRAPH and TORULES, the functions that convert a rule into a hypergraph and a tree decomposition into a rule decomposition, respectively. Nevertheless, in order to better take advantage from the features of  $\mathcal{J}$ -DLV and do not interfere with its existing optimizations, we designed ad-hoc versions for such functions.

For instance,  $\mathcal{J}$ -DLV supports the whole ASP-Core-2 language, which contains advanced constructs like aggregates, choice rules and queries; our implementation, even if resembling the one of *lpopt*, introduces custom extensions explicitly tailored to  $\mathcal{J}$ -DLV optimizations, and some updates in the way the aforementioned linguistic extensions are handled. It is worth noting that, when dealing with rules containing aggregate literals or choice atoms (Calimeri et al. 2013),  $\mathcal{J}$ -DLV rewrites them: briefly, each conjunction of literals in aggregate and/or choice elements is replaced by a fresh atom, and an auxiliary rule is added to preserve semantics; this ensures more efficiency and transparency with respect to  $\mathcal{J}$ -DLV grounding machinery and its native optimization techniques. As a result, the SMARTDECOMPOSITION algorithm, that takes place after such rewritings, does not need to explicitly take care of internal conjunctions of aggregates or choice constructs; on the contrary, *lpopt* possibly decomposes also such internal conjunctions.

Differently from *lpopt*,  $\mathcal{J}$ -DLV explicitly handles queries, and employs the magic sets rewriting technique (Alviano et al. 2012) to boost query answering; in our approach, SMARTDECOMPOSITION is applied after the magic rewriting has occurred, so that decompositions is applied also to resulting magic rules. In addition, given that  $\mathcal{J}$ -DLV performs other rewritings on the input rules for optimization purposes, the function TORULES is in charge of performing such already existing rewriting tasks also on the rules resulting from the decompositions.

Another relevant issue is related to the safety of the rules generated in a decomposition. Indeed, due to the abstract nature of SMARTDECOMPOSITION, we cannot assume that they are safe, since this depends on the schemas selected for converting a rule into a hypergraph, and a tree decomposition into a set of rules. Hence, the TORULES function must properly take this into account, as briefly noted in Section 3. In particular, our implementation, given a rule  $r$  and an associated tree decomposition  $TD$ , after a rule  $r'$  corresponding to a node in  $TD$  has been generated, checks its safety. If  $r'$  is unsafe, and  $UV$  is the set of unsafe variables in  $r'$ , an atom  $a$

over a fresh predicate  $p$ , that contains the variables in  $UV$  as terms, is added to  $B(r')$  and a new rule  $r''$  is generated, having  $a$  as head; a set of literals  $L$  binding the variables in  $UV$  is extracted from  $B(r)$  and added to  $B(r'')$ . Interestingly, the choice of the literals to be inserted in  $L$  is in general not unique, as different combinations of literals might bind the same set of variables; for instance, one might even directly add  $L$  to  $B(r')$  without generating  $r''$ ; however, this might introduce further variables in  $B(r')$ , and alter the original join operations in it. For this reason, in our implementation we decided to still add  $r''$ , and while choosing a possible binding, for each variable  $V \in UV$  we try to keep the number of literals taken from  $B(r)$  small, also preferring to pick positive literals with small ground extensions.

More in detail, given a variable  $V \in UV$ , we look for a “standard” positive literal  $l$  that binds  $V$  and features as terms only variables, constants or functional terms: the rationale behind such choice is that no additional literals will be needed to guarantee the safety of  $l$  itself. If more than one such literals exists, we select the one with the smallest extension size; if no one is available, we pick up the first suitable literal according to the following predefined priority order: classical literals featuring other kind of terms (such as arithmetic terms), built-in atoms and aggregate literals, respectively. Intuitively, for such literals, additional ones from  $B(r)$  may in turn be needed to ensure their safety. Interestingly, the choice of saviour literals is more careful than what it would be obtained by using *lpopt* as a black box, as in this case the choice could not rely on information that are available only from within the instantiation process.

The current implementation of function GENERATETREEDecompositions, which, given a hypergraph  $HG$ , is in charge of returning a set of tree decompositions  $TDS$ , relies on the open-source C++ library *htd* (Abseher et al. 2017)<sup>4</sup>, an efficient and flexible library for computing customized tree and hypertree decompositions; in our implementation, we used the most recent version available at the time of writing. The library features several methods for computing tree decompositions according to different heuristics described in literature; we took advantage from this, and our implementation allows the user to deviate from the default method via a command-line option. Interestingly, the *htd* library features also a fitness mechanism for “ranking” decompositions according to a user-provided fitness function. In our setting, we made use of such mechanism in order to associate a cost estimation relying on Formula (1) (see Section 4.1) to a computed decomposition; hence, in the decomposition selection phase,  $\mathcal{J}$ -DLV generates a number of tree decompositions and selects the best one as the one the lowest instantiation cost, according to our criteria. This constitutes another important difference w.r.t. the approach of *lpopt*, that instead, makes use of the same generation tool in order to obtain just one decomposition per each rule with no evaluation at all. Obviously, handling the fitness mechanism can imply some overhead w.r.t. the choice of computing only one, unevaluated, decomposition. For this reason, in our approach decompositions are requested and evaluated one at a time and, in order to limit the impact of such phase on performance,  $\mathcal{J}$ -DLV by default stops the generation of additional tree decompositions after 3 consecutive generations that do not show improvements in the fitness values, and never looks for more than a total of 5 generations. These limits have been set by experimentally observing that the consecutive decompositions generated by *htd* present no performance improvements with higher values; however, they can be customized by means of command-line options. The selected decomposition is then compared against the original rule in order to check whether it is convenient to actually decompose or not, as described in Section 4.1.

<sup>4</sup> <https://github.com/mabseher/htd>

An additional expedient to limit overheads consists in disabling the fitness mechanism in case of rules featuring very long bodies; indeed, computing multiple decompositions may be particularly costly for such rules (see Section 5). Therefore, we set a limit to body length so that, when it is exceeded, the fitness mechanism is automatically disabled and just one decomposition is generated and checked against the original rule. Again, we set a default value experimentally; by default, the limit is set to 10 literals per body, but it can be changed by means of a command-line option. In this respect, a possible improvement of our technique, which will be subject of future work, consists in properly generating a unique, presumably “good” enough, decomposition, thus preventing the expensive production of multiple ones.

## 5 Experimental Evaluation

We carried out a thorough experimental activity aimed at assessing the impact of SMARTDECOMPOSITION on the grounding performance of  $\mathcal{J}$ -DLV, analyzing the effectiveness of the proposed heuristics, and also at having a first glance on the effect of the produced instantiation over state-of-the-art ASP solvers. For the sake of readability, we discuss next only a significant subset of the experiments that have been carried out; additional experiments are illustrated in appendices.

### 5.1 Benchmarks and Results

All the experiments reported in this section and in the following have been performed on a NUMA machine equipped with two 2.8 GHz AMD Opteron 6320 and 128 GiB of main memory, running Linux Ubuntu 14.04.4 (kernel ver. 3.19.0-25). Binaries have been generated by the GNU C++ compiler 5.4.0. We allotted 15 GiB and 600 seconds to each system per each single run, as memory and time limits. Three versions of  $\mathcal{J}$ -DLV have been compared: (i)  $\mathcal{J}$ -DLV without any decomposition, (ii) *lpopt* (version 2.2) combined in pipeline with  $\mathcal{J}$ -DLV (i.e., a black-box usage of *lpopt*), (iii)  $\mathcal{J}$ -DLV<sup>SD</sup>, i.e.  $\mathcal{J}$ -DLV empowered with the herein introduced version of SMARTDECOMPOSITION.

As for benchmarks, we first considered the whole 6th ASP Competition suite (Gebser et al. 2015), the latest available at the time of writing; for each problem, the average time over the 20 selected instances of the official Competition runs is reported; in order to produce replicable results, the random seed used by *lpopt* for heuristics has been set to 0 for system (ii).

Results are reported in Table 1, showing number of grounded instances within the allotted time along with the average time spent. The symbol US in the table indicates that a configuration does not support the syntax of the encoding for corresponding domain; in particular, this happens in case of domains featuring queries, as system (ii) is not able to process them because of the lack of support for queries from *lpopt*.

Results of the “blind usage” of *lpopt* (system (ii)) are conflicting: for instance, in some cases it enjoys a great gain w.r.t. the version of  $\mathcal{J}$ -DLV without decomposition, in particular while dealing with the *Permutation Pattern Matching* problem, yet showing great losses in other cases, such as *Knight Tour With Holes*, where instantiating rules resulting from the decomposition requires more time w.r.t. the input ones. On the other hand, from Table 1 it is easy to see that the SMARTDECOMPOSITION algorithm allows  $\mathcal{J}$ -DLV<sup>SD</sup> to always match or overcome  $\mathcal{J}$ -DLV performances, still enjoying relevant improvements when decomposition is actually convenient (up to 96.7% in case of *Permutation Pattern Matching*), and avoiding negative effects of the

Table 1: 6th Competition (20 instances per problem) – Grounding Benchmarks: number of grounded instances and average running times (in seconds). US indicates that corresponding configurations do not support the adopted syntax.

Problem	$\mathcal{J}\text{-DLV}$		$lpopt \mid \mathcal{J}\text{-DLV}$		$\mathcal{J}\text{-DLV}^{\text{SD}}$		$\mathcal{J}\text{-DLV}^{\text{SD}}$ gap	
	#grnd	time	#grnd	time	#grnd	time	absolute	%
Abstract Dialectical Frameworks	20	0,12	20	0,12	20	0,12	0,00	0%
Combined Configuration	20	13,58	20	13,39	20	13,15	0,24	2%
Complex Optimization	20	57,56	20	60,72	20	57,24	0,32	1%
Connected Still Life	20	0,10	20	0,10	20	0,10	0,00	0%
Consistent Query Answering	20	76,44	0	US	20	77,00	-0,57	-1%
Crossing Minimization	20	0,10	20	0,10	20	0,10	0,00	0%
Graceful Graphs	20	0,30	20	0,31	20	0,30	0,00	0%
Graph Coloring	20	0,10	20	0,10	20	0,10	0,00	0%
Incremental Scheduling	20	16,07	20	15,74	20	16,21	-0,47	-3%
Knight Tour With Holes	20	1,83	20	5,98	20	1,84	-0,01	-1%
Labyrinth	20	1,97	20	1,83	20	2,02	-0,18	-10%
Maximal Clique	20	4,93	20	21,60	20	4,96	-0,03	-1%
MaxSAT	20	3,85	20	8,87	20	3,86	-0,01	0%
Minimal Diagnosis	20	5,09	20	4,30	20	4,22	0,07	2%
Nomistery	20	3,45	20	1,94	20	3,63	-1,68	-87%
Partner Units	20	0,46	20	0,47	20	0,47	0,00	0%
Permutation Pattern Matching	20	130,47	20	4,35	20	4,21	0,14	3%
Qualitative Spatial Reasoning	20	5,44	20	5,50	20	5,44	0,00	0%
Reachability	20	126,54	0	US	20	126,14	0,40	0%
Ricochet Robots	20	0,36	20	0,39	20	0,39	-0,03	-9%
Sokoban	20	1,21	20	1,23	20	1,22	-0,01	-1%
Stable Marriage	20	118,55	20	125,78	20	119,53	-0,99	-1%
Steiner Tree	20	29,00	20	28,92	20	29,11	-0,19	-1%
Strategic Companies	20	0,19	0	US	20	0,20	0,00	-1%
System Synthesis	20	1,09	20	1,15	20	1,08	0,01	1%
Valves Location Problem	20	2,52	20	2,53	20	2,54	-0,02	-1%
Video Streaming	20	0,10	20	0,10	20	0,10	0,00	0%
Visit-all	20	1,18	20	0,44	20	0,48	-0,04	-9%
Total Grounded Instances	560/560		500/560		560/560			

black-box decomposition mechanism, as in the case of *Knight Tour With Holes*. In addition, we note also that the  $\mathcal{J}\text{-DLV}^{\text{SD}}$  is able to limit the overhead w.r.t.  $\mathcal{J}\text{-DLV}$ : indeed, it is negligible even in cases where decomposition does not pay; the same does not hold for the system (ii) which suffers from the useless additional invocation of *lpopt* in all cases when the input program cannot be decomposed (see, e.g., *Maximal Clique*).

As a remark, what we expected here is that, while dealing with such benchmarks whose encodings coming from the ASP competition are already highly optimized,  $\mathcal{J}\text{-DLV}^{\text{SD}}$  performed similarly to  $\mathcal{J}\text{-DLV}$  (with no decompositions) in all cases where decomposition is not convenient, and similarly to system (ii) otherwise. In order to assess this, we computed absolute and relative differences in term of times between  $\mathcal{J}\text{-DLV}^{\text{SD}}$  and the best performing among the other two configurations, for each benchmark; data are reported in the two rightmost columns of Table 1. As it can be observed, apart from negligible fluctuations and with the only relevant excep-



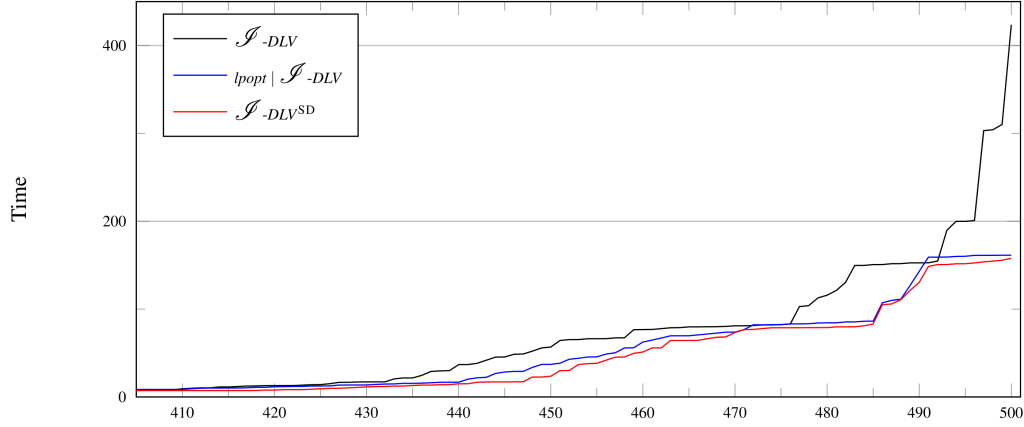


Figure 4: 6th Competition Grounding Benchmarks (excluding domains featuring queries): grounded instances over time (in seconds).

Table 2: 2QBF Grounding Benchmarks: number of total grounded instances.

$\mathcal{J}\text{-DLV}$	$lpopt \mid \mathcal{J}\text{-DLV}$	$\mathcal{J}\text{-DLV}^{\text{SD}}$
8	82	96

tion consisting of *Nomystery*, our expectations have been met: absolute differences are close to zero, meaning that  $\mathcal{J}\text{-DLV}^{\text{SD}}$  behaviour is systematically comparable with the best one among the other two. The special case of *Nomystery* is discussed later in this section.

An additional view of the general picture coming from this set of benchmarks is given by the plot in Figure 4, built over the same data of Table 1 except for the three domains featuring queries, that, as already mentioned, are unsupported by system (ii). The plot allows us to appreciate the advantage granted by the decomposition rewriting, as both systems (ii) and (iii) clearly outperform system (i), and to note that the performance reached thanks to the SMART-DECOMPOSITION algorithm are consistently better than what achieved via the unconditional use of decomposition.

Furthermore, we considered an additional set of benchmarks that have been already used in (Bichler et al. 2016b) in order to test the efficiency of ASP-solvers paired with *lpopt* over challenging programs. In particular, in (Bichler et al. 2016b), some publicly available QBF instances have been ported to ASP, according to a conversion strategy that produces programs featuring a complex structure and very long rules. This test-suite includes 200 ASP programs, each one corresponding to a different 2-QBF instance. The results are depicted in Figure 5: the number of grounded instances is on the x-axis while running times (in seconds) are on the y-axis; the total number of successfully grounded instances per each tested configuration is reported in Table 2. First of all, we note that while dealing with these problems applying a decomposition on decomposable rules is always a good choice; indeed, when no decomposition is performed the number of grounded instances is significantly smaller and running times are higher w.r.t. configurations

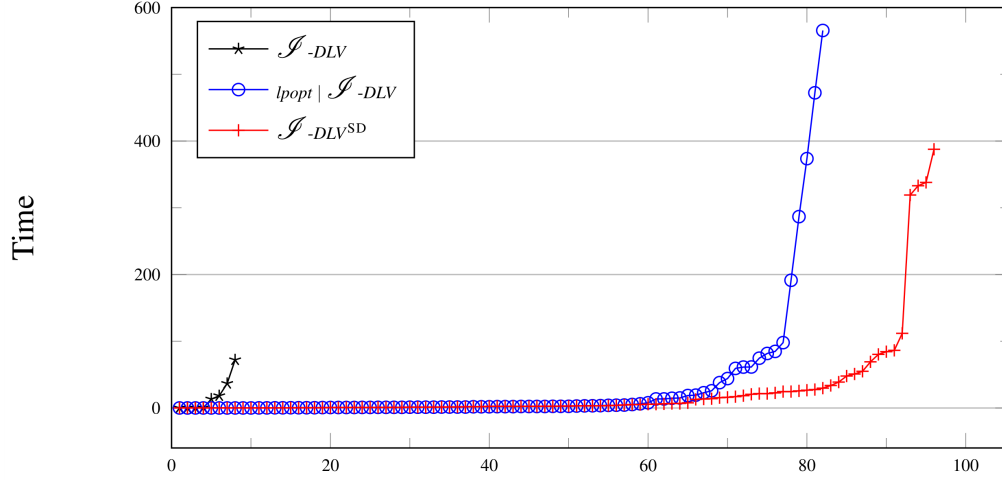


Figure 5: QBF Grounding Benchmarks: grounded instances over time (in seconds).

adopting decomposition techniques. Moreover, the heuristics guiding SMARTDECOMPOSITION in  $\mathcal{J}$ -DLV work properly, estimating the decompositions as convenient, and  $\mathcal{J}$ -DLV<sup>SD</sup> automatically disables, internally, the fitness mechanism because the body size of rules is higher than the fixed limit, thus reducing the risk of high overheads (cf. Section 4.4). In general,  $lpopt \mid \mathcal{J}$ -DLV and  $\mathcal{J}$ -DLV<sup>SD</sup> enjoy similar performance, and  $\mathcal{J}$ -DLV<sup>SD</sup> behaves as the best performing version. Although both versions decompose the same rules,  $\mathcal{J}$ -DLV<sup>SD</sup> benefits from a tight integration of the decomposition mechanism into the evaluation process that allows to better interact with the other optimization strategies of  $\mathcal{J}$ -DLV and possibly lead to different choices in decompositions. Finally, on the technical side, we note that,  $lpopt$  and  $\mathcal{J}$ -DLV<sup>SD</sup> rely on different versions of the `htd` library versions.

In summary, results of experiments clearly show the effectiveness of the herein proposed approach. Some further considerations can be done about implementation and integration into a system like  $\mathcal{J}$ -DLV. Besides merely technical aspects, it is worth remembering that, as already mentioned,  $\mathcal{J}$ -DLV is packed with a large number of optimizations; this means that a rewriting-based technique such as the SMARTDECOMPOSITION algorithm might have non-trivial interactions with them. Our experiments show that, in general, these interactions lead to performance gains, as it is clear while looking, for instance, at the QBF problem; nevertheless, a few isolated cases go towards different outcomes. In particular, looking at Table 1, we find that *Nomystery* and, even if to a smaller extent, *Labyrinth*, apparently benefit more from the black-box usage than from the heuristic-guided one. However, this is not the case: we investigated, and found that the reason is not related to the choices made according to the heuristics, but rather to the mentioned interaction with other internal rewritings performed by  $\mathcal{J}$ -DLV before the decomposition stage (for more details, we refer the reader to (Calimeri et al. 2017b)); a more detailed study of such interaction will be subject of future works.

## 5.2 On the Effectiveness of the Heuristics

In order to better understand the actual effects on grounding performance of the SMARTDECOMPOSITION algorithm as guided by the proposed heuristics, we computed some relevant statistics

Table 3: Detailed comparison of  $\mathcal{J}\text{-DLV}^{\text{SD}}$  against  $\mathcal{J}\text{-DLV}$ .

		$\mathcal{J}\text{-DLV}$	$\mathcal{J}\text{-DLV}^{\text{SD}}$	$\mathcal{J}\text{-DLV}^{\text{SD}}$ gain
All problems	#solved instances	1688	1787	6%
	Average time	22,26	15,39	31%
	# timeouts	113	19	83%
	# memouts	5	0	100%
Affected problems	# solved instances	392	491	25%
	Average time	40,10	10,28	74%
	# timeouts	103	9	91%
	# memouts	5	0	100%

starting from the data obtained from experiments over all domains considered in our experimental activities, thus including, besides those described in Section 5.1, those described in Appendices; we aggregate them over specific set of instances, as described next, and report the results in a table comparing the behaviours of  $\mathcal{J}\text{-DLV}$  and  $\mathcal{J}\text{-DLV}^{\text{SD}}$ . In particular, Table 3 shows two sets of data: the first refers to the whole collection of problem domains, while the second to the subset of “affected domains”, i.e., problems where significant differences on performance are reported, either positive or negative<sup>5</sup>. The first two columns report performance of the two system configurations, respectively, while the third reports the percentage gain achieved by  $\mathcal{J}\text{-DLV}^{\text{SD}}$  thanks to the SMARTDECOMPOSITION algorithm.

It is easy to see that the positive impact of the technique on grounding performance, on the overall (i.e., over all problems), is significant: a hundred of additional grounded instances (+6%), more than 80% of timeouts avoided, and no more instances remain unsolved because of the excessive amount of required memory. The impact is even more evident if we consider that average times are computed only over the set of instances that are solved by both  $\mathcal{J}\text{-DLV}^{\text{SD}}$  and  $\mathcal{J}\text{-DLV}$ ; still, the performance gain turns out to be over 30%.

When we focus on the set of affected problems, the benefits of the proposed techniques are even more evident; we just note here as the gain in average grounding times, still computed only over the set of instances that are grounded by both systems, is almost 75%.

### 5.3 Impact of $\mathcal{J}\text{-DLV}^{\text{SD}}$ on ASP Solvers

We proved above how a smart decomposition strategy significantly improves performance of a grounder like  $\mathcal{J}\text{-DLV}$ ; interestingly, such improvements on the instantiation process are relevant from many perspectives. First of all, as already mentioned in the introduction, a grounder like  $\mathcal{J}\text{-DLV}$  is actually a full-fledged deductive database system, that can profitably employed in many real-world domains for non-trivially querying knowledge bases of various nature, ranging from traditional relational to ontology-based ones. In these contexts, typically, programs to be evaluated turn out to be normal and stratified, and thus, completely solvable by a proper grounder. In all such cases, given that solving phase is not needed, each improvement on the grounding side

<sup>5</sup> A problem domain is here considered as “affected” if either the number of instances grounded by the two systems differ, or, in case the number is the same, difference in average grounding times between the two systems is either above +10% or below -10%.

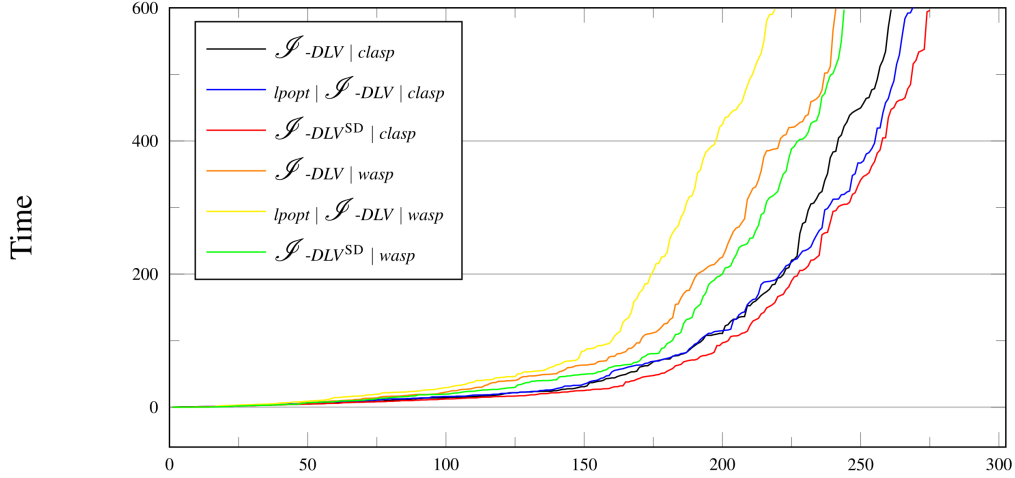


Figure 6: 6th Competition Solving Benchmarks: solved instances over time (in seconds).

trivially implies improvements on the whole ASP computation. In addition, the proposed technique can be of great help in all those cases where, given the nature of standard ASP evaluation strategy, the ground program can be so huge that it constitutes a bottleneck. The SMARTDECOMPOSITION algorithm can be useful for mitigating this issue, allowing to actually instantiate programs that cannot be grounded without: let us think, for instance, of the 2-QBF domain discussed in Section 5.1. Furthermore, even if the proposed technique aims at improving grounding, it has a positive impact also on solving times, thus allowing to improve performance of the whole computational process. To evaluate such impact we performed an additional experimental analysis; in particular, we combined the same three versions of  $\mathcal{J}$ -DLV used in Section 5.1 with the two mainstream ASP solvers *clasp* (Gebser et al. 2015) (version 5.2.1) and *wasp* (Alviano et al. 2015) (version 2.1), and tested the 6 resulting configurations over the 6th ASP Competition benchmarks.

Average times and number of solved instances within the allotted time are reported in Table 4, where time outs and cases of unsupported syntax are denoted by T0 and US, respectively. First of all, we observe that both solvers, when coupled with  $\mathcal{J}$ -DLV<sup>SD</sup>, show, in general, improved performance and solve a larger number of instances, w.r.t. the configurations with  $\mathcal{J}$ -DLV; on the contrary, the “blind usage” of *lpopt* leads, in general, to a loss of performance for both solvers: in spite of the gain in some cases, the total number of solved instances within the suite is significantly lower. A different perspective of the results is provided by Figure 6, showing solved instances over time (in seconds), where the benefits of the proposed techniques are very clear for both tested solvers.

Improvements on the overall ASP computation observable when  $\mathcal{J}$ -DLV<sup>SD</sup> is used are not only caused by improvements on grounding times; indeed, there are also cases in which solving times get better even if there is no evident gain in grounding times. This is due to the fact that the rewriting causes changes in the “form” and the size of the generated instantiation, thus often inducing positive effects on the solving side. Furthermore, it can be observed that on a same domain the effects of the decomposition on the two solvers are different: in some cases, benefits enjoyed by a solver are not reported for the other one (see, for instance, *Incremental Scheduling*). This suggests that the heuristics guiding the smart decomposition herein proposed, that already

Table 4: 6th Competition (20 instances per problem) – Solving Benchmarks: number of solved instances and average running times (in seconds). Time out and unsupported syntax issues are denoted by TO and US, respectively.

Problem	$\mathcal{J}$ -DLV   clasp		$lpopt$   $\mathcal{J}$ -DLV   clasp		$\mathcal{J}$ -DLV <sup>sd</sup>   clasp		$\mathcal{J}$ -DLV   wasp		$lpopt$   $\mathcal{J}$ -DLV   wasp		$\mathcal{J}$ -DLV <sup>sd</sup>   wasp	
	#solved	time	#solved	time	#solved	time	#solved	time	#solved	time	#solved	time
Abstract Dialectical Frameworks	20	6.88	20	7.36	20	6.89	11	33.26	11	22.38	11	32.21
Combined Configuration	8	148.64	9	176.67	10	182.41	1	311.89	0	TO	0	TO
Complex Optimization	18	149.84	19	167.58	18	149.44	6	150.30	5	99.05	6	148.00
Connected Still Life	6	220.70	6	243.05	6	222.12	12	55.02	12	78.03	12	55.47
Consistent Query Answering	20	87.05	0	US	20	87.39	18	87.47	0	US	18	88.05
Crossing Minimization	7	53.02	6	64.23	7	56.79	19	3.50	19	2.40	19	5.58
Graceful Graphs	9	141.77	10	130.97	9	140.44	6	174.17	4	122.14	6	171.42
Graph Coloring	15	162.25	15	171.39	15	160.44	8	133.71	7	217.23	8	133.34
Incremental Scheduling	13	90.62	11	39.16	14	128.81	8	155.20	5	137.74	6	124.89
Knight Tour With Holes	11	55.76	10	26.90	11	56.04	10	35.50	8	63.97	10	35.67
Labyrinth	12	63.19	11	120.09	12	67.25	11	104.73	10	168.65	11	106.71
Maximal Clique	0	TO	0	TO	0	TO	9	353.03	9	353.63	9	352.56
MaxSAT	7	39.67	7	46.81	7	39.77	19	91.01	19	95.82	19	91.49
Minimal Diagnosis	20	8.90	20	8.46	20	8.32	20	30.77	20	29.38	20	25.95
Nomystery	8	138.64	9	103.16	7	203.32	8	37.32	9	33.34	7	167.59
Partner Units	14	19.81	14	20.29	14	20.26	5	116.99	10	168.07	10	168.24
Permutation Pattern Matching	11	164.63	17	152.47	20	15.62	20	182.53	10	279.70	20	23.36
Qualitative Spatial Reasoning	20	125.13	20	125.75	20	124.97	13	145.50	13	145.23	13	145.67
Reachability	20	137.55	0	US	20	137.53	6	138.40	0	US	6	139.17
Ricochet Robots	9	67.84	12	109.32	12	188.07	7	206.86	8	87.95	9	134.22
Sokoban	8	73.95	9	82.45	8	76.90	8	86.00	9	64.59	8	88.25
Stable Marriage	5	389.26	7	341.43	5	387.85	7	410.46	7	427.66	7	431.15
Steiner Tree	3	243.89	3	244.89	3	242.45	1	131.66	1	131.75	1	131.80
Strategic Companies	17	119.63	0	US	17	122.24	7	31.38	0	US	7	30.95
System Synthesis	0	TO	0	TO	0	TO	0	TO	0	TO	0	TO
Valves Location Problem	16	43.09	16	26.09	16	43.05	15	40.93	15	39.27	15	41.32
Video Streaming	13	61.84	10	75.70	13	61.63	9	9.15	0	TO	9	9.03
Visit-all	8	16.90	8	15.22	8	15.21	8	62.11	8	61.28	8	60.06
Total Solved Instances	318/560		269/560		332/560		272/560		219/560		275/560	

shows a general positive impact on both mainstream solvers, could be further fine-tuned once a specific solver to be coupled to  $\mathcal{J}$ -DLV<sup>SD</sup> is chosen, by taking into account also its specific characteristics.

## 6 Conclusion

We introduced SMARTDECOMPOSITION, a novel technique for automatically optimizing ASP programs by means of decomposition-guided rewritings. The algorithm is designed to be adapted to different ASP implementations; furthermore, it can be customized with heuristics of choice for discerning among possible decompositions for each input rule, and determining whether applying the selected decomposition appears to be actually a “smart” choice.

In addition, we embedded a version of SMARTDECOMPOSITION in the ASP system *DLV*, and in particular in its grounding module  $\mathcal{J}$ -*DLV*. We introduced heuristics criteria for selecting decompositions that consider not only the non-ground structure of the program at hand, but also the instance it is coupled to. We experimentally tested our approach, and results are very promising: the proposed technique improves grounding performance, and highlights a positive impact, in general, also on the solving side. This is confirmed also by the results of the 7th ASP Competition (Gebser et al. 2017): here the winner was a system combining the version of  $\mathcal{J}$ -*DLV* implementing the preliminary decomposition rewriting described in (Calimeri et al. 2017a) with an automatic solver selector (Fuscà et al. 2017), that inductively chooses the best solver depending on some inherent features of the instantiation produced.

The  $\mathcal{J}$ -*DLV* system incorporating the technique herein described can be downloaded from <https://github.com/DeMaCS-UNICAL/I-DLV/wiki>, where a user guide is also reported addressing, among others, the options related to the techniques described in the present work.

As future work, we plan to investigate on further strategies for generating decompositions, starting from a more fine-grained analysis along existing ones. For instance, we note that current decompositions tend to split up a rule as much as possible, and in some cases this might require fresh predicates featuring significantly large extensions that could have a noticeable impact on performance; hence, given a set of bags composing a tree decomposition, one could check whether collapsing some bags produces some benefits with this respect. In addition, we also plan to take advantage from automatic and more advanced methods, such as machine learning mechanisms, in order to better tailor decomposition criteria and threshold values to the scenario at hand. Furthermore, we want to design a version of SMARTDECOMPOSITION specifically geared towards solvers, with the aim of further automatically optimizing the whole ASP computational process. A starting point to this direction can be the recent work of (Bliem et al. 2017), where it emerged that the performance of modern solvers are highly influenced by the tree-width of the input program; thus, this represents a starting point to explore the potential of our technique on the solving step.

## Acknowledgements

This work has been partially supported by the Italian region Calabria under project “DLV Large Scale” (CUP J28C17000220006) POR Calabria FESR 2014–2020 and by both the European Union and the Italian Ministry of Economic Development under the project EU H2020 PON I&C 2014–2020 “Smarter Solutions in the Big Data World – S2BDW” (CUP B28I17000250008).

## References

- ABSEHER, M., MUSLIU, N., AND WOLTRAN, S. 2017. htd - A free, open-source framework for (customized) tree decompositions and beyond. In *Integration of AI and OR Techniques in Constraint Programming - 14th International Conference, CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings*, D. Salvagnin and M. Lombardi, Eds. Lecture Notes in Computer Science, vol. 10335. Springer, 376–386.
- ALVIANO, M., CALIMERI, F., DODARO, C., FUSCÀ, D., LEONE, N., PERRI, S., RICCA, F., VELTRI, P., AND ZANGARI, J. 2017. The ASP system DLV2. In *Logic Programming and Nonmonotonic Reasoning - 14th International Conference, LPNMR 2017, Espoo, Finland, July 3-6, 2017, Proceedings*, M. Balducini and T. Janhunen, Eds. Lecture Notes in Computer Science, vol. 10377. Springer, 215–221.
- ALVIANO, M., DODARO, C., LEONE, N., AND RICCA, F. 2015. Advances in WASP. In *Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27-30, 2015. Proceedings*, F. Calimeri, G. Ianni, and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 9345. Springer, 40–54.
- ALVIANO, M., FABER, W., GRECO, G., AND LEONE, N. 2012. Magic sets for disjunctive datalog programs. *Artificial Intelligence* 187, 156–192.
- BEN-ELIYAHU, R. AND DECHTER, R. 1994. Propositional semantics for disjunctive logic programs. *Ann. Math. Artif. Intell.* 12, 1-2, 53–87.
- BEN-ELIYAHU-ZOHARY, R. AND PALOPOLI, L. 1997. Reasoning with minimal models: Efficient algorithms and applications. *Artif. Intell.* 96, 2, 421–449.
- BICHLER, M., MORAK, M., AND WOLTRAN, S. 2016a. lpopt: A rule optimization tool for answer set programming. In *Logic-Based Program Synthesis and Transformation - 26th International Symposium, LOPSTR 2016, Edinburgh, UK, September 6-8, 2016, Revised Selected Papers*, M. V. Hermenegildo and P. López-García, Eds. Lecture Notes in Computer Science, vol. 10184. Springer, 114–130.
- BICHLER, M., MORAK, M., AND WOLTRAN, S. 2016b. The power of non-ground rules in answer set programming. *Theory and Practice of Logic Programming* 16, 5-6, 552–569.
- BLIEM, B., MOLDOVAN, M., MORAK, M., AND WOLTRAN, S. 2017. The impact of treewidth on ASP grounding and solving. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, C. Sierra, Ed. ijcai.org, 852–858.
- BREWKA, G., EITER, T., AND TRUSZCZYNSKI, M. 2011. Answer set programming at a glance. *Communications of the ACM* 54, 12, 92–103.
- CALIMERI, F., FABER, W., GEBSER, M., IANNI, G., KAMINSKI, R., KRENNWALLNER, T., LEONE, N., RICCA, F., AND SCHAUB, T. 2013. ASP-Core-2: 4th ASP Competition Official Input Language Format. <https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.01c.pdf>.
- CALIMERI, F., FUSCÀ, D., PERRI, S., AND ZANGARI, J. 2017a. The ASP instantiator I-DLV. In *PAoASP*. Espoo, Finland.
- CALIMERI, F., FUSCÀ, D., PERRI, S., AND ZANGARI, J. 2017b. I-DLV: the new intelligent grounder of DLV. *Intelligenza Artificiale* 11, 1, 5–20.
- CALIMERI, F., GEBSER, M., MARATEA, M., AND RICCA, F. 2016. Design and results of the fifth answer set programming competition. *Artificial Intelligence* 231, 151–181.
- CALIMERI, F., PERRI, S., AND RICCA, F. 2008. Experimenting with parallelism for the instantiation of ASP programs. *J. Algorithms* 63, 1-3, 34–54.
- DANTSIN, E., EITER, T., GOTTLÖB, G., AND VORONKOV, A. 2001. Complexity and expressive power of logic programming. *ACM Computing Surveys* 33, 3, 374–425.
- DAO-TRAN, M., EITER, T., FINK, M., WEIDINGER, G., AND WEINZIERL, A. 2012. Omiga : An open minded grounding on-the-fly answer set solver. In *Logics in Artificial Intelligence - 13th European Conference, JELIA 2012, Toulouse, France, September 26-28, 2012. Proceedings*, L. F. del Cerro, A. Herzig, and J. Mengin, Eds. Lecture Notes in Computer Science, vol. 7519. Springer, 480–483.
- EITER, T., GOTTLÖB, G., AND MANNILA, H. 1997. Disjunctive datalog. *ACM Transactions on Database Systems* 22, 3, 364–418.

- EITER, T., KAMINSKI, T., AND WEINZIERL, A. 2017. Lazy-grounding for answer set programs with external source access. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, C. Sierra, Ed. ijcai.org, 1015–1022.
- FABER, W., LEONE, N., AND PERRI, S. 2012. The intelligent grounder of DLV. In *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, 247–264.
- FUSCÀ, D., CALIMERI, F., ZANGARI, J., AND PERRI, S. 2017. I-DLV+MS: preliminary report on an automatic ASP solver selector. In *Proceedings of the 24th RCRA International Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion 2017 co-located with the 16th International Conference of the Italian Association for Artificial Intelligence (AI\*IA 2017), Bari, Italy, November 14-15, 2017.*, M. Maratea and I. Serina, Eds. CEUR Workshop Proceedings, vol. 2011. CEUR-WS.org, 26–32.
- GEBSER, M., KAMINSKI, R., KAUFMANN, B., ROMERO, J., AND SCHAUB, T. 2015. Progress in clasp series 3. In *Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27-30, 2015. Proceedings*, F. Calimeri, G. Ianni, and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 9345. Springer, 368–383.
- GEBSER, M., KAMINSKI, R., KAUFMANN, B., AND SCHAUB, T. 2014. Clingo = ASP + control: Preliminary report. *CoRR abs/1405.3694*.
- GEBSER, M., KAMINSKI, R., KÖNIG, A., AND SCHAUB, T. 2011. Advances in gringo series 3. In *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, J. P. Delgrande and W. Faber, Eds. Lecture Notes in Computer Science, vol. 6645. Springer, 345–351.
- GEBSER, M., KAUFMANN, B., AND SCHAUB, T. 2012. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence* 187, 52–89.
- GEBSER, M., MARATEA, M., AND RICCA, F. 2015. The design of the sixth answer set programming competition - - report -. In *Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27-30, 2015. Proceedings*, F. Calimeri, G. Ianni, and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 9345. Springer, 531–544.
- GEBSER, M., MARATEA, M., AND RICCA, F. 2016. What’s hot in the answer set programming competition. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, D. Schuurmans and M. P. Wellman, Eds. AAAI Press, 4327–4329.
- GEBSER, M., MARATEA, M., AND RICCA, F. 2017. The design of the seventh answer set programming competition. In *Logic Programming and Nonmonotonic Reasoning - 14th International Conference, LPNMR 2017, Espoo, Finland, July 3-6, 2017, Proceedings*, M. Balduccini and T. Janhunen, Eds. Lecture Notes in Computer Science, vol. 10377. Springer, 3–9.
- GELFOND, M. AND LIFSCHITZ, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing* 9, 3/4, 365–386.
- GIUNCHIGLIA, E., LIERLER, Y., AND MARATEA, M. 2006. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning* 36, 4, 345–377.
- GOTTLÖB, G., GRECO, G., LEONE, N., AND SCARCELLO, F. 2016. Hypertree decompositions: Questions and answers. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, T. Milo and W. Tan, Eds. ACM, 57–74.
- GOTTLÖB, G., GROHE, M., MUSLIU, N., SAMER, M., AND SCARCELLO, F. 2005. Hypertree decompositions: Structure, algorithms, and applications. In *Graph-Theoretic Concepts in Computer Science, 31st International Workshop, WG 2005, Metz, France, June 23-25, 2005, Revised Selected Papers*, D. Kratsch, Ed. Lecture Notes in Computer Science, vol. 3787. Springer, 1–15.
- GOTTLÖB, G., LEONE, N., AND SCARCELLO, F. 2001. Hypertree decompositions: A survey. In *Mathematical Foundations of Computer Science 2001, 26th International Symposium, MFCS 2001 Mariánské Lázně, Czech Republic, August 27-31, 2001, Proceedings*, J. Sgall, A. Pultr, and P. Kolman, Eds. Lecture Notes in Computer Science, vol. 2136. Springer, 37–57.
- JANHUNEN, T., NIEMELÄ, I., SEIPEL, D., SIMONS, P., AND YOU, J. 2006. Unfolding partiality and disjunctions in stable model semantics. *ACM Transactions on Computational Logic* 7, 1, 1–37.



- KAUFMANN, B., LEONE, N., PERRI, S., AND SCHAUB, T. 2016. Grounding and solving in answer set programming. *AI Magazine* 37, 3, 25–32.
- LEFÈVRE, C., BÉATRIX, C., STÉPHAN, I., AND GARCIA, L. 2017. Asperix, a first-order forward chaining approach for answer set computing. *Theory and Practice of Logic Programming* 17, 3, 266–310.
- LEONE, N., PERRI, S., AND SCARCELLO, F. 2001. Improving ASP instantiators by join-ordering methods. In *Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001, Vienna, Austria, September 17-19, 2001, Proceedings*, T. Eiter, W. Faber, and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 2173. Springer, 280–294.
- LEONE, N., PFEIFER, G., FABER, W., EITER, T., GOTTLÖB, G., PERRI, S., AND SCARCELLO, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic* 7, 3, 499–562.
- LIFSCHITZ, V. 1999. Answer set planning. In *Logic Programming: The 1999 International Conference, Las Cruces, New Mexico, USA, November 29 - December 4, 1999*, D. D. Schreye, Ed. MIT Press, 23–37.
- MORAK, M. AND WOLTRAN, S. 2012. Preprocessing of complex non-ground rules in answer set programming. In *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4-8, 2012, Budapest, Hungary*, A. Dovier and V. S. Costa, Eds. LIPIcs, vol. 17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 247–258.
- PALÙ, A. D., DOVIER, A., PONTELLI, E., AND ROSSI, G. 2009. GASP: answer set programming with lazy grounding. *Fundamenta Informaticae* 96, 3, 297–322.
- PERRI, S., RICCA, F., AND SIRIANNI, M. 2013. Parallel instantiation of ASP programs: techniques and experiments. *TPLP* 13, 2, 253–278.
- PERRI, S., SCARCELLO, F., CATALANO, G., AND LEONE, N. 2007. Enhancing DLV instantiator by backjumping techniques. *Ann. Math. Artif. Intell.* 51, 2-4, 195–228.
- ROBERTSON, N. AND SEYMOUR, P. D. 1986. Graph minors. II. algorithmic aspects of tree-width. *J. Algorithms* 7, 3, 309–322.
- SIMONS, P., NIEMELÄ, I., AND SOININEN, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence* 138, 1-2, 181–234.
- SYRJÄNEN, T. 2001. Omega-restricted logic programs. In *Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001, Vienna, Austria, September 17-19, 2001, Proceedings*, T. Eiter, W. Faber, and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 2173. Springer, 267–279.
- ULLMAN, J. D. 1988. *Principles of Database and Knowledge-Base Systems, Volume I*. Principles of computer science series, vol. 14. Computer Science Press.
- WARD, J. AND SCHLIPF, J. S. 2004. Answer set programming with clause learning. In *Logic Programming and Nonmonotonic Reasoning, 7th International Conference, LPNMR 2004, Fort Lauderdale, FL, USA, January 6-8, 2004, Proceedings*, V. Lifschitz and I. Niemelä, Eds. Lecture Notes in Computer Science, vol. 2923. Springer, 302–313.
- WEINZIERL, A. 2017. Blending lazy-grounding and CDNL search for answer-set solving. In *Logic Programming and Nonmonotonic Reasoning - 14th International Conference, LPNMR 2017, Espoo, Finland, July 3-6, 2017, Proceedings*, M. Balduccini and T. Janhunen, Eds. Lecture Notes in Computer Science, vol. 10377. Springer, 191–204.

## Appendix A Experiments on Automatic Optimization

In Section 5 we discussed the results of tests over the benchmark suite from the 6th ASP Competition. It is worth noting that many domains have been included in several subsequent editions of the ASP Competition series; over the years, the participant teams have iteratively fine-tuned the encodings with the aim of maximizing performance of competing ASP systems. This led, in the case of 6th Competition, to a bunch of programs that are already optimized for ASP computation, thus limiting the room for further improvements.

Table A 1: 4th Competition – Grounding Benchmarks: number of grounded instances and average running times (in seconds). US indicates that corresponding configurations do not support the adopted syntax.

Problem	#inst.	<i><math>\mathcal{J}</math>-DLV</i>		<i>lpopt</i>   <i><math>\mathcal{J}</math>-DLV</i>		<i><math>\mathcal{J}</math>-DLV<sup>SD</sup></i>	
		#grounded	time	#grounded	time	#grounded	time
Abstract Dialectical Frameworks	30	30	0.13	30	0.13	30	0.13
Bottle Filling Problem	30	30	4.12	30	6.86	30	4.39
Chemical Classification	30	30	87.81	30	403.38	30	88.22
Complex Optimization *	29	29	36.28	29	38.39	29	36.07
Connected Still Life *	10	10	0.12	10	0.13	10	0.15
Crossing Minimization *	30	30	0.10	30	0.10	30	0.10
Graceful Graphs	30	30	0.37	30	0.39	30	0.37
Graph Colouring *	30	30	0.10	30	0.10	30	0.10
Hanoi Tower	30	30	0.22	30	0.23	30	0.30
Incremental Scheduling *	30	12	297.95	17	229.49	21	221.17
Knight Tour with Holes *	30	20	176.99	20	181.16	20	178.59
Labyrinth	30	30	1.49	30	1.40	30	1.51
Maximal Clique *	30	30	0.34	30	1.11	30	0.34
Minimal Diagnosis *	30	30	2.54	30	2.20	30	2.57
Nomystery *	30	30	34.91	21	100.14	30	35.28
Permut. Pattern Matching *	30	28	57.71	30	3.64	30	62.32
Qualitative Spatial Reasoning *	30	30	2.85	30	2.87	30	2.84
Reachability	30	30	101.93	0	US	30	102.04
Ricochet Robots	30	30	0.27	30	0.31	30	0.31
Sokoban	30	30	2.65	30	2.68	30	2.69
Solitaire	27	27	0.13	27	0.18	27	0.20
Stable Marriage *	30	30	28.35	30	2.65	30	2.46
Strategic Companies	30	30	0.19	0	US	30	0.19
Valves Location	30	30	3.97	30	3.98	30	3.93
Visit-all *	30	30	0.13	30	0.14	30	0.13
Weighted-Sequence Problem *	30	30	2.87	30	9.61	30	2.95
Total Grounded Instances		726/756		664/756		737/756	

On the one hand, such considerations strengthen the positive results reported in Section 5; on the other hand, one might wonder about what is the impact of the SMARTDECOMPOSITION algorithm when dealing with less refined encodings. We find such inquiry of interest, as it should be in the spirit of the declarative nature of ASP to allow developers to concentrate on knowledge representation rather than on low-level performance issues, that also lead, in some cases, to the production of encodings rather involved and less “human readable”. This is why, in addition to the experiments reported in Section 5, we tested our approach also on benchmarks coming from older editions of the ASP Competition series. In particular, we considered the 4th Competition, as it is the farthest in time in which encodings comply to the ASP-Core-2 input language standard.

In this set of experiments we measured grounding times of the same three versions of  *$\mathcal{J}$ -DLV* already taken into account in Section 5, within the same experimental environment (hardware, software, memory and time limits). Table A 1 shows the results; problem names reported in italic denotes domains which are in common between the 4th and the 6th Competition, while

Table A 2: Variation of the encodings – Grounding Benchmarks: number of grounded instances and average running times (in seconds).

Problem	#inst.	$\mathcal{J}\text{-DLV}$		$\mathcal{J}\text{-DLV}^{\text{SD}}$					
		#grounded	time	#grounded	time	#grounded	time	#grounded	time
4th Competition Instances									
		4th Comp. Enc.		6th Comp. Enc.		4th Comp. Enc.		6th Comp. Enc.	
Incr. Scheduling	30	12	297.95	30	54.65	21	221.17	30	1.93
Maximal Clique	30	30	0.34	30	2.96	30	0.34	30	3.11
Minimal Diagnosis	30	30	2.54	30	1.76	30	2.57	30	1.76
Nomystery	30	30	34.91	30	47.24	30	35.28	30	47.11
Perm. Pattern Match.	30	28	57.71	30	0.27	30	62.32	30	0.27
Stable Marriage	30	30	28.35	30	3.16	30	2.46	30	2.86
6th Competition Instances									
		4th Comp. Enc.		6th Comp. Enc.		4th Comp. Enc.		6th Comp. Enc.	
Incr. Scheduling	20	11	336.77	20	16.07	19	211.61	20	16.21
Maximal Clique	20	20	6.63	20	4.93	20	6.58	20	4.96
Minimal Diagnosis	20	20	4.12	20	5.09	20	4.14	20	4.22
Nomystery	20	20	55.11	20	3.45	20	43.74	20	3.63
Perm. Pattern Match.	20	16	168.93	20	130.47	20	150.99	20	4.21
Stable Marriage	20	0	TO	20	118.55	20	172.68	20	119.53

those marked with a '\*' symbol feature more optimized encodings in the 6th. As expected, in this scenario the positive impact of SMARTDECOMPOSITION is even more evident:  $\mathcal{J}\text{-DLV}^{\text{SD}}$  grounds a larger number of instances in a significant smaller average time. Intuitively, the less an encoding is fine-tuned, the highest are likely to be the benefits stemming from a careful automatic rewriting of input rules.

Furthermore, for all problems in common between the two ASP competitions herein considered, we tested the systems over the programs obtained by coupling the encodings featured by the 4th with the instances featured by the 6th, and vice-versa. This should provide us with further information about the impact of both “manual” optimizations and the ones coming from our automatic method. Intuitively, an encoding may be optimized in different ways, and not necessarily by means of a syntactic modification of rules; for instance, one can push additional information about the domain at hand into the encoding, possibly with constraints, in order to reduce the search space. The results are reported in Table A 2. It is clear that, as one can expect, the best combination is given by  $\mathcal{J}\text{-DLV}^{\text{SD}}$  fed with optimized encoding coming from 6th Competition. However, some interesting considerations can be made: indeed, in several cases (for instance, *Permutation Pattern Matching*, both over instances from the 4th, and, even more, over instances from the 6th, that appear to be harder), the smart decomposition guarantees similar or better performance improvements in grounding times that are obtained by the manual tuning.

Table B 1: Additional Grounding Benchmarks: number of grounded instances and average running times (in seconds).

Problem	#inst.	$\mathcal{J}\text{-DLV}$		$lpopt \mid \mathcal{J}\text{-DLV}$		$\mathcal{J}\text{-DLV}^{\text{SD}}$	
		#grounded	time	#grounded	time	#grounded	time
Cutedge	130	130	34.38	130	1.64	130	1.08
Graph 5col	180	180	0.12	180	0.14	180	0.12
Ground Explosion 2	17	7	73.33	7	73.02	7	72.87
Reach	50	50	0.29	50	0.76	50	0.30
TimeTabling	27	27	85.57	27	63.67	27	57.67
Total Solved Instances		<b>367/377</b>		<b>367/377</b>		<b>367/377</b>	

## Appendix B Additional Experiments

We report next the results of an additional experimental evaluation over a further set of benchmark. We take into account problem domains that have been already used for assessing performance of ASP systems in other works; in particular, we considered the domains *Cutedge*, *Graph 5col*, *Ground Explosion 2*, *Reach* (Weinzierl 2017) and *TimeTabling* (Perri et al. 2007; Calimeri et al. 2008; Perri et al. 2013).

Table B 1 reports grounding times of the same three versions of  $\mathcal{J}\text{-DLV}$  already taken into account in Section 5, within the same experimental environment (hardware, software, memory and time limits). We first note that, regarding *Reach*, even if the problem domain is the same as *Reachability* of Section 5, both encoding and instances are different, as they are taken from (Weinzierl 2017), and do not feature queries. In this case, where decomposition is not applicable because of the rule structure, results show that the use of SMARTDECOMPOSITION, as previously noted, allows us to avoid the overhead due to the invocation of *lpopt*. On the overall, again, one can observe the positive impact of SMARTDECOMPOSITION over grounding times, that allows significant performance improvements in case of *Cutedge* and *TimeTabling*, where  $\mathcal{J}\text{-DLV}^{\text{SD}}$  times are 97% and 33% lower, respectively, w.r.t.  $\mathcal{J}\text{-DLV}$ .