TPLP **24** (1): 132–156, 2024.

doi:10.1017/S1471068423000078 First published online 22 June 2023

System Predictor: Grounding Size Estimator for Logic Programs under Answer Set Semantics[*]

DANIEL BRESNAHAN, NICHOLAS HIPPEN and YULIYA LIERLER
University of Nebraska Omaha, Omaha, NE, USA
(*e-mails:* daniel.b.bresnahan@gmail.com, nhippen@unomaha.edu, ylierler@unomaha.edu)

submitted 31 August 2022; revised 17 January 2023; accepted 31 March 2023

Abstract

Answer set programming is a declarative logic programming paradigm geared towards solving difficult combinatorial search problems. While different logic programs can encode the same problem, their performance may vary significantly. It is not always easy to identify which version of the program performs the best. We present the system PREDICTOR (and its algorithmic backend) for estimating the grounding size of programs, a metric that can influence a performance of a system processing a program. We evaluate the impact of PREDICTOR when used as a guide for rewritings produced by the answer set programming rewriting tools PROJECTOR and LPOPT. The results demonstrate potential to this approach.

KEYWORDS: answer set programming, encoding optimizations

1 Introduction

Answer set programming (ASP) (Brewka *et al.* 2011) is a declarative (constraint) programming paradigm geared towards solving difficult combinatorial search problems. ASP programs model problem specifications/constraints as a set of logic rules. These logic rules define a problem instance to be solved. An ASP system is then used to compute solutions (answer sets) to the program. Answer set programming has been successfully used in scientific and industrial applications. Examples include, but are not limited to a decision support systems for space shuttle flight controllers (Balduccini *et al.* 2006), team building and scheduling (Ricca *et al.* 2012), and healthcare realm (Dodaro *et al.* 2021).

Intuitive ASP encodings are not always the most optimal/performant, making this programming paradigm less attractive to novice users as their first attempts to problem solving may not scale. ASP programs often require careful design and expert knowledge in order to achieve performant results (Gebser *et al.* 2011a). Figure 1 depicts a typical ASP system architecture. The first step performed by systems called grounders transforms a non-ground logic program (with variables) into a ground/propositional program (without

[*] We would like to thank Mirek Truszczynski, Daniel Houston, Liu Liu, Michael Dingess, Roland Kaminski, Abhishek Parakh, Victor Winter, Parvathi Chundi, and Jorge Fandinno for valuable discussions on the subject of this paper. The work was partially supported by NSF grant 1707371.

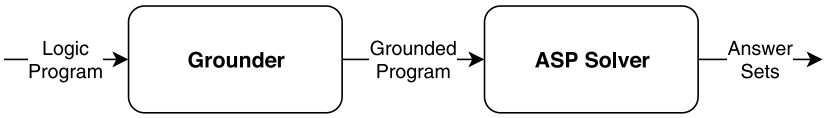

Fig. 1. Typical ASP system architecture.

variables). Expert ASP programmers often modify their ASP solution targeting the reduction of grounding size of a resulting program. Size of a ground program has been shown to be a predictive factor of a program's performance, enabling it to be used as an "optimization metric" (Gebser *et al.* 2011a). Intelligent grounding techniques (Faber *et al.* 2012) utilized by grounders such as GRINGO (Gebser *et al.* 2011b) or IDLV (Calimeri *et al.* 2017) also keep such a reduction in mind. Intelligent grounding procedures analyze a given (non-ground) program to produce a smaller propositional program without altering the solutions. In addition, researchers looked into automatic program rewriting procedures. Systems such as SIMPLIFY (Eiter *et al.* (2006a); Eiter *et al.* (2006b)), LPOPT (Bichler (2015); Bichler *et al.* (2020)), and PROJECTOR (Hippen and Lierler 2019) rewrite non-ground programs (preserving their semantics) targeting the reduction of the grounding size. These systems are meant to be prepossessing tools agnostic to the later choice of ASP solving technology. Tools such as SIMPLIFY, LPOPT, and PROJECTOR, despite illustrating promising results, often hinder their objective. Sometimes, the original set of rules is better than the rewritten set, when their size of grounding and/or runtime is taken as a metric. Research has been performed to mitigate the negative impact of these rewritings. For example, Mastria *et al.* (2020) demonstrated a novel approach to guide automatic rewriting techniques performed in IDLV using machine learning with a set of features built from structural properties of a considered program and domain information. Thus, a machine learning model guides IDLV on whether to perform built-in rewritings or not. Another example of incorporating automatic rewriting techniques with the use of information about specifics of a considered program and a considered grounder is work by Calimeri *et al.* (2019). In that work, the authors incorporated program rewriting technique stemming from LPOPT into the intelligent grounding algorithm of grounder IDLV. Such tight coupling of the rewriting and grounding procedures allows IDLV to make a decision on whether to apply or not an LPOPT rewriting based on the current state of grounding. Grounder IDLV accurately estimates the impact of rewriting on grounding and based on this information decides whether to perform a rewriting. This synergy of intelligent grounding and a rewriting technique demonstrates the best performant results. Yet, it makes the transfer of rewriting techniques laborious assuming the need of tight integration of any rewriting within a grounder of choice. *Here*, we propose an algorithm for estimating the size of grounding a program based on (i) mimicking an intelligent grounding procedure documented by Faber *et al.* (2012) and (ii) techniques used in query optimization in relational databases, see, for instance, Chapter 13 by Silberschatz *et al.* (1997). We then implement this algorithm in a system called PREDICTOR. This tool is meant to be used as a decision support mechanism for ASP program rewriting systems so that they perform a possible rewriting based on estimates produced by PREDICTOR. This work culminates in the integration of PREDICTOR within the rewriting tools PROJECTOR and LPOPT, which then are used prior to the invocation of a typical

https://doi.org/10.1017/S1471068423000078 Published online by Cambridge University Press

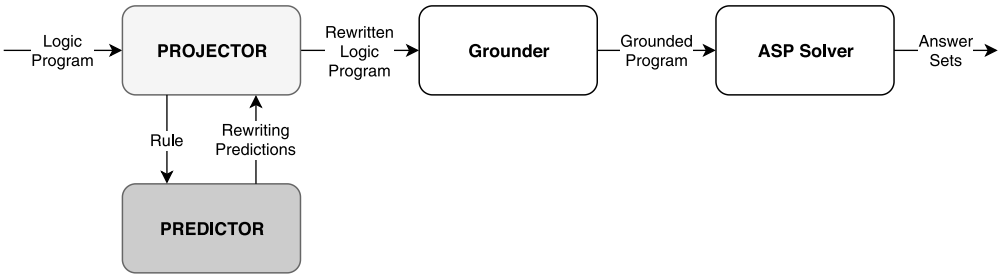

Fig. 2. An ASP system with PROJECTOR using PREDICTOR.

grounder-solver pair of ASP. For example, Figure 2 depicts the use of PREDICTOR within the rewriting system PROJECTOR as a preprocessing step before the invocation of an ASP system. To depict the use of PREDICTOR within the rewriting system LPOPT as a preprocessing step it is sufficient to replace the box named PROJECTOR by a box named LPOPT in Figure 2. We illustrate the success of this synergy by an experimental analysis. It is due to note that PREDICTOR is a stand-alone tool and can be used as part of any ASP inspired technology where its functionality is of interest.

We underline that the important contribution of this work is in the design of a building block – in the shape of the system PREDICTOR – towards making ASP a truly declarative framework. Answer set programming is frequently portrayed as a powerful declarative programming formalism. Yet, we can argue that such a claim is somewhat misleading. At present, to achieve scalable ASP solutions to problems of interest, it is typical that *an expert ASP programmer* – with strong insights into underlying grounding/solving technology – constructs logic programs/encodings for problems that are efficient rather than intuitive. The ASP experts must rely on their extensive knowledge of the ASP technology to deliver efficient solutions. Yet, in truly declarative formalism we would expect the possibility of constructing *intuitive* encodings and rely on underlying systems to process these efficiently. This way programmers may focus on coding specifications of problems at hand rather than the specifics of the shape of these specifications and the details of the underlying technology. This paper targets the development of infrastructure, which *one day* will allow us to achieve the ultimate goal of *truly declarative ASP*. Ultimately, an expert ASP programmer capable of devising efficient encodings will be replaced by an ASP user capable of devising intuitive specifications that are then turned into effective specification by a portfolio of automatic tools such as, for example, PROJECTOR and PREDICTOR, or LPOPT and PREDICTOR pairs showcased and evaluated here in the final section of the paper. This work makes a step towards achieving the described ultimate goal: it provides us with insights and possible directions for the developments on that pass.

Related work. It is due to remark on another body of research that targets a similar goal namely portfolio-like approaches, where researchers use machine learning based methods in navigating the space of distinct ASP grounders and/or solvers – CLASPFOLIO (Hoos *et al.* 2014); ME-ASP (Maratea *et al.* 2014); or encodings – ESP (Liu *et al.* 2022) to decide on the best possibility in tackling considered problem by means of ASP technology. All and all, to the best of our knowledge this work is *one of the very few* approaches for the stated/similar purpose. Already mentioned work by Mastria *et al.* (2020) presents an alternative machine learning based method for a similar purpose. In

https://doi.org/10.1017/S1471068423000078 Published online by Cambridge University Press

that work properties of a program are considered to predict whether rewriting will help an ASP solver down the road or not. Also, the work by Calimeri *et al.* (2019) can be seen as the most related one to this paper. The greatest difference of the championed approach is its detachment from any specific grounding system. It produces its estimates looking at a program alone. Calimeri *et al.* incorporate computation of estimates within a grounder. The benefit of such approach that at any point in time their estimates are reflective of de facto grounding that happened so far.

Outline of the paper. We start by introducing the subject matter terminology. The key contribution of the work lies in the development of formulas for estimating the grounding size of a logic program based on its structural analysis and insights on intelligent grounding procedures. First, we present the simplified version of these formulas for the case of tight programs. We trust that this helps the reader to build intuitions for the work. Second, the formulas for non-tight programs are given. We then describe the implementation details of system PREDICTOR. The main part of the presentation concerns most typical logic rules (stemming from Prolog). The section that follows the presentation of the key concepts discusses other kinds of rules and their treatment by the PREDICTOR system. We conclude by experimental evaluation that includes incorporation of PREDICTOR within rewriting systems PROJECTOR and LPOPT.

Parts of this paper appeared in the proceedings of the 17th Edition of the European Conference on Logics in Artificial Intelligence (Hippen and Lierler 2021).

2 Preliminaries

An *atom* is an expression $p(t_1, ..., t_k)$, where p is a predicate symbol of arity $k \geq 0$ and $t_1, ..., t_k$ are *terms* – either object constants or variables. As customary in logic programming, variables are marked by an identifier starting with a capital letter. We assume object constants to be numbers. This is an inessential restriction as we can map strings to numbers using, for instance, the lexicographic order. For example, within our implementation described in this paper: we consider all alphanumeric object constants occurring in a program; sort these object constants using the lexicographic order; and map each string in this sorted list to a natural number that corresponds to its position in the list added to the greatest natural number occurring in the program.

For an atom $p(t_1, ..., t_k)$ and position i $(1 \leq i \leq k)$, we define an *argument* denoted by $p[i]$. By $p(t_1, ..., t_k)^0$ and $p(t_1, ..., t_k)^i$ we refer to predicate symbol p and the term t_i, respectively. A *rule* is an expression of the form

$$a_0 \leftarrow a_1, ..., a_m, not\ a_{m+1}, ..., not\ a_n. \tag{1}$$

where $n \geq m \geq 0$, a_0 is either an atom or symbol $\bot$, and $a_1, ..., a_n$ are atoms. We refer to a_0 as the *head* of the rule and an expression to the right hand side of an arrow symbol in (1) as the *body*. An atom a and its negation *not* a is a *literal*. To literals $a_1, ..., a_m$ in the body of rule (1) we refer as *positive*, whereas to literals $not\ a_{m+1}, ..., not\ a_n$ we refer as *negative*. For a rule r, by $\mathbb{H}(r)$ we denote the head atom of r. By $\mathbb{B}^+(r)$ we denote the set of positive literals in the body of r. We obtain the set of variables present in an atom a and a rule r by $vars(a)$ and $vars(r)$, respectively. For a variable X occurring in rule r, by $args(r, X)$ we denote the set

https://doi.org/10.1017/S1471068423000078 Published online by Cambridge University Press

$$\{p[i] \mid a \in \mathbb{B}^+(r), a^0 = p, \text{ and } a^i = X\}.$$

In other words, $args(r, X)$ denotes the set of arguments in the positive literals of rule r, where variable X appears. A rule r is *safe* if each variable in r appears in $\mathbb{B}^+(r)$. Let r be a safe rule

$$p(A) \leftarrow q(A, B), r(1, A), not\ s(B). \tag{2}$$

Then $vars(r) = \{A, B\}$, $args(r, A) = \{q[1], r[2]\}$, and $args(r, B) = \{q[2]\}$. A *(logic) program* is a finite set of safe rules. We call programs containing variables *non-ground*.

For a program Π, $oc(p[i])$ denotes the set of all object constants occurring within

$$\{\mathbb{H}(r)^i \mid r \in \Pi \text{ and } \mathbb{H}(r)^0 = p\},$$

whereas $oc(\Pi)$ denotes the set of all object constants occurring in the head atoms of the rules in Π.

Example 2.1
Let Π_1 denote a program

$$p(1).\ p(2).\ r(3). \tag{3}$$
$$q(X, 1) \leftarrow p(X). \tag{4}$$

Then, $oc(p[1]) = \{1, 2\}$, $oc(q[1]) = \emptyset$, $oc(q[2]) = \{1\}$ and $oc(\Pi_1) = \{1, 2, 3\}$. The *grounding* of a program Π, denoted $gr(\Pi)$, is a ground program obtained by instantiating variables in Π with all object constants of the program. For example, $gr(\Pi_1)$ consists of rules in (3) and rules

$$q(1, 1) \leftarrow p(1). \quad q(2, 1) \leftarrow p(2). \tag{5}$$
$$q(3, 1) \leftarrow p(3). \tag{6}$$

Given a program Π, ASP grounders utilizing intelligent grounding are often able to produce a program smaller than its grounding $gr(\Pi)$, but that has the same answer sets as $gr(\Pi)$. Recall program Π_1 introduces in Example 2.1. For instance, the program obtained from $gr(\Pi_1)$ by dropping rule (6) may be a result of intelligent grounding. The *ground extensions* of a predicate within a grounded program Π are the set of terms associated with the predicate in the program. For instance, in $gr(\Pi_1)$, the ground extensions of predicate q is the set of tuples $\{\langle 1, 1\rangle, \langle 2, 1\rangle, \langle 3, 1\rangle\}$. For an argument $p[i]$ and a ground program Π, we call the number of distinct object constants occurring in the ground extensions of p in Π at position i the *argument size* of $p[i]$. For instance, for program $gr(\Pi_1)$ argument sizes of $p[1]$, $q[1]$, and $q[2]$ are 3, 3, and 1, respectively.

The *dependency graph* of a program Π is a directed graph $G_\Pi = \langle N, E\rangle$ such that N is the set of predicates appearing in Π and E contains the edge (p, q) if there is a rule r in Π in which p occurs in $\mathbb{B}^+(r)$ and q occurs in the head of r. A program Π is *tight* if G_Π is acyclic, otherwise the program is *non-tight* (Fages 1994).

Example 2.2
Let Π_2 denote a program constructed from Π_1 (introduced in Example 2.1) by extending it with rules:

$$r(2).\ r(4). \tag{7}$$
$$s(X, Y, Z) \leftarrow r(X), p(X), p(Y), q(Y, Z). \tag{8}$$

https://doi.org/10.1017/S1471068423000078 Published online by Cambridge University Press

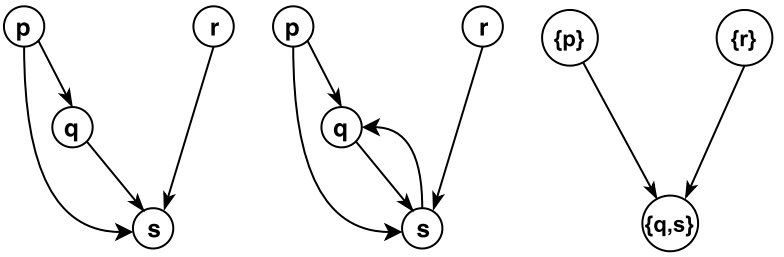

Fig. 3. Left: Graph G_{Π_2}; Center: Graph G_{Π_3}; Right: Graph $G^{sc}_{\Pi_3}$.

Program Π_3 is the program Π_2 extended with the rule:

$$q(Y, X) \leftarrow s(X, Y, Z). \tag{9}$$

Figure 3 shows the dependency graphs G_{Π_2} (left) and G_{Π_3} (center). Program Π_2 is tight, while program Π_3 is not.

3 System PREDICTOR

The key contribution of this work is the development of the system PREDICTOR (its algorithmic and software base), whose goal is to provide estimates for the size of an "intelligently" grounded program. In other words, its goal is to assess the impact of grounding without grounding itself. PREDICTOR is based on the intelligent grounding procedures implemented by the grounder DLV, described in Faber *et al.* (2012). The key difference is that, instead of building the ground instances of each rule in the program, PREDICTOR constructs statistics about the predicates, their arguments, and rules of the program. This section provides formulas we developed in order to produce the estimates backing up the computed statistics. We conclude with details on the implementation.

It is due to make couple remarks. First, in a way we parallel the work on query optimization techniques within relational databases, for example, see Chapter 13 in Silberschatz *et al.* (1997). Indeed, when a particular query is considered within a relational database there are often numerous ways to its execution/implementation. Relational databases maintain statistics about its tables to produce estimates for intermediate results of various execution scenarios of potential queries. These estimates help database management systems decide which of the possible execution plans of the query at hand to select. In this work, we develop methods to collect and maintain statistics/estimates about entities of answer set programs. We then show how these estimates may help a rewriting (prepossessing) system for ASP to decide whether to rewrite some rules of a program or not.

Second, the intelligent grounding procedure implemented by grounder DLV (Faber *et al.* 2012) is based on database evaluation techniques (Ullman 1988; Abiteboul *et al.* 1995). The same statement is the case for another modern grounder GRINGO (Gebser *et al.* 2011b; Kaminski and Schaub 2022). It also shares a lot in common with grounder DLV. This fact makes the estimates of system PREDICTOR rooting in the algorithm of DLV applicable also within the framework of GRINGO. In a nutshell, both DLV and GRINGO instantiate a program via an iterative bottom-up process starting from the program's facts targeting the accumulation of ground atoms and ground rules derivable from the rules seen so far. As this process continues, a new ground rule is produced when its

https://doi.org/10.1017/S1471068423000078 Published online by Cambridge University Press

positive body atoms belong to the already computed atoms. Then, the head atom of this rule is added to the set of already accumulated ground atoms. This process continues until no new ground atoms/rules are produced by this process.

Argument size estimation. Tight program case: The estimation formulas are based on predicting argument sizes. To understand these it is essential to describe an order in which we produce estimates for predicate symbols/arguments. Given a program Π, we obtain such an ordering by performing a topological sorting on its dependency graph. We associate each node in this ordering with its position and call it a *strata rank* of a predicate. For example, p, q, r, s is one possible ordering for program Π_2 (introduced in Example 2.2). This ordering associates strata ranks $1, 2, 3, 4$ with predicates p, q, r, s, respectively.

We now introduce some intermediate formulas for constraining our estimates. These intermediate formulas are inspired by query optimization techniques within relational databases, for example, see Chapter 13 in Silberschatz *et al.* (1997). These formulas keep track of information that helps us to estimate which actual values may occur in the grounded program without storing these values themselves. Let $p[i]$ be an argument. We track the range of values that may occur at this argument. To provide intuitions for an introduced process, consider an intelligent grounding of Π_2 consisting of rules (3), (5), (7), and rules

$$s(2,1,1) \leftarrow r(2), p(2), p(1), q(1,1). \tag{10}$$

$$s(2,1,1) \leftarrow r(2), p(2), p(2), q(2,1). \tag{11}$$

This intelligent grounding produces rules (10), (11) in place of rule (8). Variable X from rule (8) is only ever replaced with object constant 2. Intuitively, this is due to the intersection $oc(p[1]) \cap oc(r[1]) = \{2\}$. We model such a restriction by considering what minimum and maximum values are possible for each argument in an intelligently grounded program (compliant with described principle; all modern intelligent grounders respect such a restriction). We then use these values to define an "upper restriction" of the argument size for each argument.

For a tight program Π, let $p[i]$ be an argument in Π; R be the following set of rules

$$\{r \mid r \in \Pi,\ \mathbb{H}(r)^0 = p, \text{ and } \mathbb{H}(r)^i \text{ is a variable}\}. \tag{12}$$

By $\downarrow^{t\text{-}t}_{est}(p[i])$ we denote an estimate of a minimum value that may appear in argument $p[i]$ in Π:

$$\downarrow^{t\text{-}t}_{est}(p[i]) = min\big(oc(p[i]) \cup \{max\Big(\{\downarrow^{t\text{-}t}_{est}(p'[i']) \mid p'[i'] \in args(r, \mathbb{H}(r)^i)\}\Big) \mid r \in R\}\big).$$

The superscript t-t stands for "tight." Note how $\mathbb{H}(r)^i$ in $args(r, \mathbb{H}(r)^i)$ is conditioned to be a variable due to the choice of set R of rules. The function $\downarrow^{t\text{-}t}_{est}$ is total because the rank of the predicate occurring on the left hand side of the definition above is strictly greater than the ranks of all of the predicate symbols p' on the right hand side, where rank is understood as a strata rank defined before (multiple strata rankings are possible; any can be considered here). By $\uparrow^{t\text{-}t}_{est}(p[i])$ we denote an estimate of a maximum value that may appear in argument $p[i]$ in tight program Π. It is computed using formula for $\downarrow^{t\text{-}t}_{est}(p[i])$ with min, max, and $\downarrow^{t\text{-}t}_{est}$ replaced by max, min, and $\uparrow^{t\text{-}t}_{est}$, respectively.

https://doi.org/10.1017/S1471068423000078 Published online by Cambridge University Press

Now that we have estimates for minimum and maximum values, we estimate the size of the range of possible values. We understand the *range* of an argument to be the number of values we anticipate to see in the argument within an intelligently grounded program if the values were all integers between the minimum and maximum estimates. It is possible that our minimum estimate for a given argument is greater than its maximum estimate. Intuitively, this indicates that no ground rule will contain this argument in its head. The number of values between the minimum and maximum estimates may also be greater than the number of object constants in a considered program. In this case, we restrict the range to the number of object constants occurring in the program. We compute the range, $range^{t\text{-}t}_{est}(p[i])$, as follows:

$$min\big(\{max(\{0, \uparrow^{t\text{-}t}_{est}(p[i]) - \downarrow^{t\text{-}t}_{est}(p[i]) + 1\}), |oc(\Pi)|\}\big).$$

Example 3.1
Recall program Π_2 introduced in Example 2.2. The operations required to compute the minimum estimate for argument $s[1]$ in Π_2 follow:

$$\begin{aligned}
&\downarrow^{t\text{-}t}_{est}(r[1]) = min\big(oc(r[1])\big) = 2\\
&\downarrow^{t\text{-}t}_{est}(p[1]) = min\big(oc(p[1])\big) = 1\\
&\downarrow^{t\text{-}t}_{est}(s[1]) = min(oc(s[1]) \cup\\
&\quad \{max\big(\{\downarrow^{t\text{-}t}_{est}(r[1]), \downarrow^{t\text{-}t}_{est}(p[1])\}\big)\}) = min(\emptyset \cup \{2\}) = 2.
\end{aligned}$$

We compute $\uparrow^{t\text{-}t}_{est}(s[1])$ to be 2. Then, $range^{t\text{-}t}_{est}(s[1])$ is

$$\begin{aligned}
&min(\{max\big(\{0, \uparrow^{t\text{-}t}_{est}(s[1]) - \downarrow^{t\text{-}t}_{est}(s[1]) + 1\}\big), |oc(\Pi_2)|\})\\
&= min(\{max\big(\{0, 2 - 2 + 1\}\big), 4\}) = 1.
\end{aligned}$$

We presented formulas for estimating the range of values in program's arguments. We now show how these estimates are used to assess the *size* of an argument understood as the number of distinct values occurring in this argument upon an intelligent grounding. We now outline intuitions behind a recursive process that we capture in formulas. Let $p[i]$ be an argument. If $p[i]$ is such that predicate p has no incoming edges in the program's dependency graph, then we estimate the size of $p[i]$ as $|oc(p[i])|$. Otherwise, consider rule r such that $\mathbb{H}(r)^0 = p$ and $\mathbb{H}(r)^i$ is a variable. Our goal is to estimate the *number of values* variable $\mathbb{H}(r)^i$ may be replaced with during intelligent grounding. To do so, we consider the argument size estimates for arguments in the positive body of the rule that contain variable $\mathbb{H}(r)^i$. Based on typical intelligent grounding procedures, variable $\mathbb{H}(r)^i$ may not take more values than the minimum of those argument size estimations. This gives us an estimate of the argument size relative to a single rule r. The argument size estimate of $p[i]$ with respect to the entire program may be then computed as the sum of such estimates for all rules such as r (recall that rule r satisfies the requirements $\mathbb{H}(r)^0 = p$ and $\mathbb{H}(r)^i$ is a variable). Yet, the sum over all rules may heavily overestimate the argument size. To lessen the effect of overestimation we incorporate range estimates discussed before into the described computations.

For a tight program Π, let $p[i]$ be an argument in Π; R be the set (12) of rules. By $S^{t\text{-}t}_{est}(p[i])$ we denote an estimate of the argument size $p[i]$ in Π. This estimate is computed

https://doi.org/10.1017/S1471068423000078 Published online by Cambridge University Press

as follows:

$$S^{t\text{-}t}_{est}(p[i]) = min\Big(\Big\{range^{t\text{-}t}_{est}(p[i]),\ |oc(p[i])| + \sum_{r\in R} min\big(\{S^{t\text{-}t}_{est}(p'[i']) \mid p'[i'] \in args(r, \mathbb{H}(r)^i)\}\big)\Big\}\Big).$$

We can argue that the function $S^{t\text{-}t}_{est}$ is total in the same way as we argued that the function $\downarrow^{t\text{-}t}_{est}$ is total.

Example 3.2
Let us illustrate the computation of the argument size estimates for argument $s[2]$ in program Π_2 (introduced in Example 2.2). Given that $range^{t\text{-}t}_{est}(s[2]) = 2$ and $oc(s[2]) = \emptyset$:

$$\begin{aligned}
&S^{t\text{-}t}_{est}(p[1]) = |oc(p[1])| = 2\\
&S^{t\text{-}t}_{est}(q[1]) = min(range^{t\text{-}t}_{est}(q[1]), \{|oc(q[1])| +\\
&\quad min\big(\{S^{t\text{-}t}_{est}(p[1])\}\big)\}) = min(\{2, 0 + min(\{2\})\}) = 2\\
&S^{t\text{-}t}_{est}(s[2]) = min\big(range^{t\text{-}t}_{est}(s[2]),\\
&\quad \big\{|oc(s[2])| + min\big(\{S^{t\text{-}t}_{est}(p[1]), S^{t\text{-}t}_{est}(q[1])\}\big)\big\}\big) = 2.
\end{aligned}$$

Arbitrary (non-tight) program case: To process arbitrary programs (tight and non-tight), we must manage the circular dependencies such as present in sample program Π_3 defined in Example 2.2 in the section on preliminaries. We borrow and simplify a concept of the component graph by Faber *et al.* (2012). The *component graph* of a program Π is an acyclic directed graph $G^{sc}_{\Pi} = \langle N, E\rangle$ such that N is the set of strongly connected components in the dependency graph G_{Π} of Π and E contains the arc (P, Q) if there is an arc (p, q) in G_{Π} where $p \in P$ and $q \in Q$. For tight programs, we identify its component graph with the dependency graph itself by associating a singleton set annotating a node with its member. Figure 3 (right) shows the component graph for program Π_3. For a program Π, we obtain an ordering on its predicates by performing a topological sorting on its component graph. We associate each node in this ordering with its position and call it a *strong strata rank* of each predicate that belongs to a node. For example, $\{p\}, \{r\}, \{q, s\}$ is one possible topological sorting of $G^{sc}_{\Pi_3}$. This ordering associates the following strong strata ranks $1, 2, 3, 3$ with predicates p, r, q, s, respectively.

Let C be a node/component in graph G^{sc}_{Π}. By $\mathscr{P}_C$ we denote the set

$$\{r \mid p \in C, r \in \Pi, \text{ and } \mathbb{H}(r)^0 = p\}.$$

We call this set a *module*. A rule r in module $\mathscr{P}_C$ is a *recursive rule* if there exists an atom a in the positive body of r so that $a^0 = p$ and predicate p occurs in C. Otherwise, rule r is an *exit rule*. For tight programs, all rules are exit rules. It is also possible to have modules with only recursive rules.

Example 3.3
The modules in program Π_3 introduced in Example 2.2 contain

$$\mathscr{P}_{\{p\}} = \{p(1).\quad p(2).\};\quad \mathscr{P}_{\{r\}} = \{r(2).\quad r(3).\quad r(4).\};$$

and $\mathscr{P}_{\{q,s\}}$ composed of rules (4), (8), and (9). The rules (8) and (9) are recursive.

https://doi.org/10.1017/S1471068423000078 Published online by Cambridge University Press

In the sequel we consider components whose module contains an exit rule. For a component C and its module $\mathscr{P}_C$, $M_1, ..., M_n$ $(n \geq 1)$ in the following way: Every exit rule of $\mathscr{P}_C$ is a member of M_1. A recursive rule r in $\mathscr{P}_C$ is a member of M_k $(k > 1)$ if

- for every predicate $p \in C$ occurring in $\mathbb{B}^+(r)$, there is a rule r' in $M_1 \cup ... \cup M_{k-1}$, where $\mathbb{H}(r')^0 = p$ and
- there is a predicate q occurring in $\mathbb{B}^+(r)$ such that there is a rule r'' in M_{k-1}, where $\mathbb{H}(r'')^0 = q$.

We refer to the unique partition created in this manner as the *component partition* of C; integer n is called its *cardinality*. We call elements of a component partition *groups* (the component partition is undefined for components whose module does not contain an exit rule). Prior to illustrating these concepts by an example we introduce one more notation. For a component partition $M_1, \ldots, M_k, \ldots, M_n$, by $M_k^{p[i]}$ we denote the set

$$\{r \mid r \in M_k,\ \mathbb{H}(r)^0 = p, \text{ and } \mathbb{H}(r)^i \text{ is a variable}\};$$

and by $M_{1...k}^{p[i]}$ we denote the union $\bigcup_{j=1}^{k} M_j^{p[i]}$.

Example 3.4
Recall program Π_3 from Example 2.2. The component partition of node $\{q, s\}$ in $G_{\Pi_3}^{sc}$ follows:

$$M_1 = \{q(X, 1) \leftarrow p(X).\}$$
$$M_2 = \{s(X, Y, Z) \leftarrow r(X), p(X), p(Y), q(Y, Z).\}$$
$$M_3 = \{q(Y, X) \leftarrow s(X, Y, Z).\}.$$

For program Π_3 and its argument $q[1]$:

$$M_{1...3}^{q[1]} = \{q(X, 1) \leftarrow p(X). \quad q(Y, X) \leftarrow s(X, Y, Z).\}$$

We now generalize range and argument size estimation formulas for tight programs to the case of arbitrary programs. These formulas are more complex than their "tight versions," yet they perform similar operations at their core. Intuitively, formulas for tight programs rely on argument ordering provided by the program's dependency graph. Now, in addition to an order provided by the component dependency graph, we rely on the orders given to us by the component partitions of the program.

In the remainder of this section, let Π be a program; $p[i]$ be an argument in Π; C be the node in the component graph of Π so that $p \in C$; n be the cardinality of the component partition of C; and j be an integer such that $1 \leq j \leq n$.

If the module of C does not contain an exit rule, then the estimate of the range of an argument $p[i]$, denoted $range_{est}(p[i])$, is assumed 0 and the estimate of the size of an argument $p[i]$, denoted $S_{est}(p[i])$, is assumed 0.

We now consider the case when the module of C contains an exit rule. By $\downarrow_{est}(p[i])$ we denote an estimate of a minimum value that may appear in argument $p[i]$ in program Π:

$$\downarrow_{est}(p[i]) = \downarrow_{est}^{gr}(p[i], n)$$
$$\downarrow_{est}^{gr}(p[i], j) = min(oc(p[i]) \cup \{\downarrow_{est}^{rule}(p[i], j, r) \mid r \in M_{1...j}^{p[i]}\})$$
$$\downarrow_{est}^{rule}(p[i], j, r) = max\big(\{\downarrow_{est}^{split}(p[i], p'[i'], j) \mid p'[i'] \in args(r, \mathbb{H}(r)^i)\}\big)$$
$$\downarrow_{est}^{split}(p[i], p'[i'], j) = \begin{cases} \downarrow_{est}^{gr}(p'[i'], j-1), & \text{if } p' \text{ in the same component as } p \\ \downarrow_{est}(p'[i']), & \text{otherwise} \end{cases}.$$

https://doi.org/10.1017/S1471068423000078 Published online by Cambridge University Press

We note the strong similarity between the combined definitions of $\downarrow^{gr}_{est}(p[i], j)$ and $\downarrow^{rule}_{est}(p[i], j, r)$ compared to the corresponding "tight" formula $\downarrow^{t\text{-}t}_{est}(p[i])$. Formula for $\downarrow^{split}_{est}(p[i], p'[i'], j)$ serves two purposes. If the predicate p' is in the same component as predicate p, we decrement the counter j (intuitively bringing us to preceding groups in component partition). Otherwise, we simply use the minimum estimate for $p'[i']$ that is due to the computation relevant to another component.

We now show that defined functions $\downarrow_{est}$, $\downarrow^{gr}_{est}$, $\downarrow^{rule}_{est}$ and $\downarrow^{split}_{est}$ are total. Consider any strong strata ranking of program's predicates. Then, by $rank(p)$ we refer to the corresponding strong strata rank of a predicate p. The following table provides ranks associated with expressions used to define functions in question:

Expression	Rank
$\downarrow_{est}(p[i])$	$\omega \cdot (rank(p) + 1)$
$\downarrow^{gr}_{est}(p[i], j)$	$\omega \cdot rank(p) + j$
$\downarrow^{rule}_{est}(p[i], j, r)$	$\omega \cdot rank(p) + j$
$\downarrow^{split}_{est}(p[i], p'[i'], j)$	$\omega \cdot rank(p) + j$

,

where ω is the smallest infinite ordinal number. It is easy to see that in definitions of functions $\downarrow_{est}$, $\downarrow^{gr}_{est}$, and $\downarrow^{rule}_{est}$ the ranks associated with their expressions do not increase. In definition of $\downarrow^{split}_{est}$ in terms of $\downarrow_{est}$, the rank decreases. Thus, the defined functions are total.

By $\uparrow_{est}(p[i])$ we denote an estimate of a maximum value that may appear in argument $p[i]$ in program Π. It is computed using formula for $\downarrow_{est}(p[i])$ with min, max, $\downarrow_{est}$, $\downarrow^{gr}_{est}$, $\downarrow^{rule}_{est}$, and $\downarrow^{split}_{est}$ replaced with max, min, $\uparrow_{est}$, $\uparrow^{gr}_{est}$, $\uparrow^{rule}_{est}$, and $\uparrow^{split}_{est}$, respectively. The range of an argument $p[i]$, denoted $range_{est}(p[i])$, is computed by the formula of $range^{t\text{-}t}_{est}(p[i])$, where we replace $\downarrow^{t\text{-}t}_{est}$ and $\uparrow^{t\text{-}t}_{est}$ with $\downarrow_{est}$ and $\uparrow_{est}$, respectively.

We define the formula for finding the argument size estimates, $S_{est}(p[i])$, as follows:

$$S_{est}(p[i]) = S^{gr}_{est}(p[i], n)$$

$$S^{gr}_{est}(p[i], j) = min\Big(\big\{range_{est}(p[i]), |oc(p[i])| + \sum_{r \in M^{p[i]}_{1\ldots j}} S^{rule}_{est}(p[i], j, r)\big\}\Big)$$

$$S^{rule}_{est}(p[i], j, r) = \ min\Big(\big\{S^{split}_{est}(p[i], p'[i'], j) \mid p'[i'] \in args(r, \mathbb{H}(r)^i)\big\}\Big)$$

$$S^{split}_{est}(p[i], p'[i'], j) = \begin{cases} S^{gr}_{est}(p'[i'], j-1), & \text{if } p' \text{ is in the same component as } p \\ S_{est}(p'[i']), & \text{otherwise} \end{cases}.$$

We can argue that the function S_{est} is total in the same way as we argued that the function $\downarrow_{est}$ is total.

Program size estimation. Keys We borrow the concept of a key from relational databases. This concept allows us to produce more accurate final estimates as it carries important structural information about predicates and the kinds of instantiations possible for them. (Table 1 presented in the section on experimental analysis illustrates the impact of information on the keys within the implemented system.) For some predicate p, we refer to any set of arguments of p that can uniquely identify all ground extensions of p as a *superkey* of p. We call a minimal superkey a *candidate key*. For instance, let the following be the ground extensions of some predicate q:

https://doi.org/10.1017/S1471068423000078 Published online by Cambridge University Press

$$\{\langle 1,1,a\rangle, \langle 1,2,b\rangle, \langle 1,3,b\rangle, \langle 2,1,c\rangle, \langle 2,2,c\rangle, \langle 2,3,a\rangle\}.$$

It is easy to see that both $\{q[1], q[2]\}$ and $\{q[1], q[2], q[3]\}$ are superkeys of q, while $\{q[1]\}$ is not a superkey. Only superkey $\{q[1], q[2]\}$ is a candidate key. A *primary key* of a predicate p is a single chosen candidate key. A predicate may have at most one primary key. For the purposes of this work, we allow the users of PREDICTOR to manually specify the primary key. It is possible that some predicates do not have primary keys specified. To handle such predicates, we define $key(p)$ to mean the following:

$$key(p) = \begin{cases} \text{the primary key of } p, & \text{if } p \text{ has a primary key specified} \\ \{p[1], ..., p[n]\}, & \text{otherwise} \end{cases},$$

where n is the arity of p. We call an argument $p[i]$ a *key argument* if it is in $key(p)$. For a rule r, by $kvars(r)$ we denote the set of its variables that occur in its key arguments.

Rule size estimation We now have all the ingredients to provide an estimate for grounding size of each rule in a program. We understand a *grounding size* of a rule as the number of rules produced as a result of intelligently grounding this rule. For a rule r in a program Π, the estimated grounding size, denoted $S_{est}(r)$, is computed as follows:

$$S_{est}(r) = \prod_{X \in kvars(r)} min\big(\{S_{est}(p[i]) \mid p[i] \in args(r, X)\}\big).$$

Implementation details. System PREDICTOR[1] is developed using the Python 3 programming language. PREDICTOR utilizes PYCLINGO version 5, a Python API sub-system of answer set solving toolkit CLINGO (Gebser *et al.* 2015). The PYCLINGO API enables users to easily access and enhance ASP processing steps within Python code, including access to some data in the processing chain. In particular, PREDICTOR uses PYCLINGO to parse a logic program into an abstract syntax tree (AST) representation. After obtaining the AST, PREDICTOR has an immediate access to internal rule structure of the program and computes estimates for the program using the presented formulas. System PREDICTOR is designed for integration with other systems processing ASP programs. It is distributed as a package that can be imported into other systems developed in Python 3, or it can be accessed through a command line interface. In order to ensure that system PREDICTOR is applicable to real world problems, it supports ASP-Core-2 logic programs. For instance, the estimation formulas presented here generalize well to programs with choice rules and disjunction. Rules with aggregates are also supported. Yet, for such rules more sophisticated approaches are required to be more precise at estimations. Next section covers key details on the ASP-Core-2 support by the PREDICTOR system. We then conclude by integrating the PREDICTOR system into two rewriting tools, namely, PROJECTOR and LPOPT. We present a thorough experimental analysis for these systems and the enhancement that PREDICTOR offers to them.

[1] https://www.unomaha.edu/college-of-information-science-and-technology/natural-language-processing-and-knowledge-representation-lab/software/predictor.php

https://doi.org/10.1017/S1471068423000078 Published online by Cambridge University Press

4 Language extensions: ASP-Core-2 Support

In order to ensure that system PREDICTOR is applicable to real world problems, it has been designed to operate on many common features of ASP-Core-2 logic programs. In the following we extend the definition of logic rules to include these features and discuss how these features are handled by PREDICTOR.

Pools and intervals. In ASP-Core-2 logic programs, an atom may have the form $p(t_1; ...; t_n)$, where p is a predicate of arity 1, and $t_1; ...; t_n$ is a semicolon separated list of terms. Here, $t_1; ...; t_n$ is a *pool* term. A predicate with a pool term is "syntactic sugar" that indicates there is a copy of that rule for every object constant in the pool.

Example 4.1
The following rule containing pool terms:

$$p(a; b) \leftarrow q(c; d).$$

can be expanded to the following rules:

$$p(a) \leftarrow q(c).$$
$$p(a) \leftarrow q(d).$$
$$p(b) \leftarrow q(c).$$
$$p(b) \leftarrow q(d).$$

Similarly, ASP-Core-2 programs may contain atoms of the form $p(l..r)$, where p is a predicate of arity 1, and l, r are terms. Here, $l..r$ is an *interval* term. A predicate with an interval term is "syntactic sugar" indicating that there is a copy of this rule for every integer between the range of l to r, inclusive.

Example 4.2
The following rule containing interval terms:

$$p(1..3, a) \leftarrow q(1..2).$$

can be expanded to the following rules:

$$p(1, a) \leftarrow q(1).$$
$$p(1, a) \leftarrow q(2).$$
$$p(2, a) \leftarrow q(1).$$
$$p(2, a) \leftarrow q(2).$$
$$p(3, a) \leftarrow q(1).$$
$$p(3, a) \leftarrow q(2).$$

For both pool and interval terms, system PREDICTOR handles the program as though it were in its expanded form.

Aggregates. An *aggregate element* has the form

$$t_0, ..., t_k : a_0, ..., a_m, not\ a_{m+1}, ..., not\ a_n.$$

where $k \geq 0$, $n \geq m \geq 0$, $t_0, ..., t_k$ are terms and $a_0, ..., a_n$ are atoms. An *aggregate atom* has the form

https://doi.org/10.1017/S1471068423000078 Published online by Cambridge University Press

$$\#aggr\{e_0, ..., e_n\} \prec t,$$

where $n \geq 0$ and $e_0, ..., e_n$ are aggregate elements. Symbol $\#aggr$ is either $\#count$, $\#sum$, $\#max$, or $\#min$. Symbol $projectorec$ is either $<$, $\leq$, $=$, $\neq$, $>$, or $\geq$. Symbol t is a term.

System PREDICTOR supports rules containing aggregates to a limited extent. In particular, PREDICTOR will simplify such a rule as if it had no aggregate atoms.

Example 4.3
The rule containing an aggregate atom:

$$p(X) \leftarrow q(X), \#count\{Y : r(X, Y)\} < 3.$$

is seen by PREDICTOR as the following rule:

$$p(X) \leftarrow q(X).$$

while the only variable seen in this rule will be X.

It is important to note that if an aggregate contains variables, it is possible that the *length of a rule* expands during grounding processes, where it is understood that the length of a rule is the number of atoms in a rule. We do not consider this length expansion when computing the grounding size of a rule.

Disjunctive and choice rules. A *disjunctive rule* is an extended form of ASP logic rule that allows disjunctions in its head. They are of the form

$$a_0 \vee ... \vee a_k \leftarrow a_{k+1}, ..., a_m, not\ a_{m+1}, ..., not\ a_n.$$

where $n \geq m \geq k \geq 0$, and $a_0, ..., a_n$ are atoms.

System PREDICTOR handles a disjunctive rule by replacing it with the set of rules created in the following way. For each atom a in the head of a disjunctive rule r, PREDICTOR creates a new rule of the form $a \leftarrow \mathbb{B}(r)$. For computing range and argument size estimates, all of these newly created rules are used. However, when estimating the grounding size of the original rule, only one of the rules is used.

Example 4.4
The disjunctive rule r:

$$p(1) \vee p(2) \leftarrow q(1).$$

is replaced by the following two rules:

$$p(1) \leftarrow q(1).$$
$$p(2) \leftarrow q(1).$$

Yet, only one of those rules is used for estimating the grounding size of the original rule. Using these rules is sufficient for estimating grounding information, even though they are not semantically equivalent to the original disjunctive rule.

A *condition* is of the form

$$a_0 : a_1, ..., a_m, not\ a_m + 1, ..., not\ a_n,$$

where $n \geq m \geq 0$, and $a_0, ..., a_n$ are atoms. We refer to a_0 as the head of the condition. A *choice atom* is of the form $l\{c_1; ...; c_n\}r$, where l is an integer, r is an integer such

https://doi.org/10.1017/S1471068423000078 Published online by Cambridge University Press

that $r \geq l$, and $c_1; ...; c_n$ is a semicolon separated list of conditions. We now extend the definition of a rule given by (1) to allow the head to be a choice atom. We refer to rules whose head contains a choice atom as *choice rules*.

System PREDICTOR handles a choice rule similarly to the case of a disjunctive rule, replacing it with the set of rules created in the following way. For each atom a in the head of a condition in the choice atom in rule r, create a new rule of the form $a \leftarrow \mathbb{B}(r)$. For computing range and argument size estimates, all of these newly created rules are used. However, when estimating the grounding size of the original rule, only one of the rules will be used. Note that, as with aggregates, choice rules can increase the length of a rule.

Example 4.5
The choice rule:

$$1\{p(X) : q(1); p(Y)\}1 \leftarrow r(X, Y), s(Y).$$

is replaced by the following two rules:

$$p(X) \leftarrow r(X, Y), s(Y).$$
$$p(Y) \leftarrow r(X, Y), s(Y).$$

Yet, only one of those rules is used for estimating the grounding size of the original rule.

Functions. In ASP-Core-2, a term may also be of the form $f(t_1, ..., t_n)$, where f is a function symbol and $t_1, ..., t_n$ $(n > 0)$ are term. We call terms of this form *function* terms. In order to be more compliant with ASP-Core-2 features, PREDICTOR is capable of running on programs containing function terms, however when a function term is encountered by PREDICTOR, it simply sees the function term as an object constant.

Binary operations. The ASP-Core-2 standard also allows *binary operation* terms. A binary operation term is of the form t_1 *op* t_2, where t_1 and t_2 are either an integer object constant, a variable, or a binary operation and *op* is a valid binary operator[2]. If an atom contains a binary operation term, system PREDICTOR handles it in one of three ways. If the binary operation has no variables, it treats the term as an object constant. If the binary operation contains exactly one variable, it treats the term as that variable. Otherwise, the atom is treated as if it were part of the negative body (and therefore not used in estimations).

Example 4.6
In the following rule containing binary operation terms:

$$\leftarrow p(1 + 1), q(2 * X + 1), r(2 * X + Y), s(Y).$$

the atoms are viewed as follows. Atom $p(1 + 1)$ is seen as containing an object constant term. Atom $q(2 * X + 1)$ is seen as the atom $q(X)$. Atom $r(2 * X + Y)$ is seen as being part of the negative body.

[2] http://potassco.sourceforge.net/doc/pyclingo/clingo.ast.html#BinaryOperator

https://doi.org/10.1017/S1471068423000078 Published online by Cambridge University Press

5 Experimental analysis

We investigated the utility of system PREDICTOR by integrating it as a decision support mechanism into the ASP rewriting tool PROJECTOR to create tool PRD-PROJECTOR, as well as the ASP rewriting tool LPOPT, to create tool PRD-LPOPT. These tools are discussed in following subsections.

5.1 System PRD-PROJECTOR

Figure 2 (presented in the Introduction section) demonstrates how PREDICTOR is integrated with system PROJECTOR resulting in what we call PRD-PROJECTOR. Note how PREDICTOR runs entirely independent of and prior to the grounding step of a considered ASP grounder-solver pair.

The rewriting tool PROJECTOR is documented by Hippen and Lierler (2019). This tool focuses on so called projection technique. In its default settings, it studies each rule of a given program and when a projection rewriting is applicable to considered rules PROJECTOR rewrites these accordingly. Thus, whenever the rewriting is established to be possible it is also performed. The PRD-PROJECTOR tool extends the PROJECTOR system by the decision-making mechanism supported by PREDICTOR on whether to perform rewriting or not. When PROJECTOR establishes that a rewriting is possible the system PREDICTOR evaluates an original rule against its rewritten counterpart as far as their predicted grounding sizes. The projection rewriting will only be applied if the rewritten rule is predicted to produce smaller grounding footprint. In particular, for each rule r in program Π, PROJECTOR will create a set R of rules, which represents one of the possible "projected"-versions of r. This set R of rules is then substituted into Π to create program Π'. If the predicted grounding size for this new program is smaller than, or equal to the original, the set R of rules is kept and Π' becomes a considered program in the future evaluations. However, if the new predicted grounding size is larger than the original, set R is discarded, and PRD-PROJECTOR will move on to the next rule in Π. To summarize, tool PREDICTOR is used by PROJECTOR in two ways:

1. When PRD-PROJECTOR encounters a tie through its default heuristics of PROJECTOR for selecting variables to project, PRD-PROJECTOR generates the resulting projections for each of the variables and use the projection that is predicted to have the smallest grounding size.
2. PRD-PROJECTOR only performs a projection if the prediction for the projection is smaller than the predicted grounding size for the original rule.

We note that it is possible for projections to occur inside of aggregate expressions. System PREDICTOR is not used to decide if these projections should be performed, so that these projections always occur in PRD-PROJECTOR.

5.2 System PRD-LPOPT

Figure 2 with the box representing PROJECTOR replaced by the box representing LPOPT demonstrates how PREDICTOR is integrated with system LPOPT. We refer to the version of

https://doi.org/10.1017/S1471068423000078 Published online by Cambridge University Press

LPOPT integrated with PREDICTOR as PRD-LPOPT. Once again, PREDICTOR runs entirely independent of and prior to the grounding step.

The rewriting tool LPOPT is documented by Bichler (2015); Bichler *et al.* (2020). This tool focuses on so called rule decomposition technique. This technique is strongly related to a rewriting championed by system PROJECTOR. In fact, PROJECTOR and LPOPT can be characterized as the tools performing the same kind of rewriting, while using different heuristics on how and when to apply this rewriting. Both systems attempt reducing the number of variables occurring in a rule by (a) introducing an auxiliary predicate and (b) replacing an original rule by new rules. In other words, there are often multiple ways available for rewriting the same rule and these systems may champion different ways. In its default settings, LPOPT studies each rule of a given program and when a rule decomposition rewriting is applicable to considered rules LPOPT rewrites these accordingly. Thus, it behaves just as the PROJECTOR system when used with its default settings: whenever the rewriting at hand is established to be possible it is also performed. The PRD-LPOPT tool extends the LPOPT system by the decision-making mechanism of PREDICTOR on whether to perform rewriting or not in the same manner as PRD-PROJECTOR tool extends the PROJECTOR system by the decision-making mechanism of PREDICTOR. We refer the reader to the previous subsection for the details.

5.3 Evaluation

To evaluate the usefulness of PREDICTOR, two sets of experiments are performed. First, an "intrinsic" evaluation over accuracy of the predicted grounding size compared to the actual grounding size is examined. Second, an "extrinsic" evaluation of systems PRD-PROJECTOR and PRD-LPOPT is conducted to examine whether the system PREDICTOR is indeed of use as a decision support mechanism on whether to perform or not the rewritings of PROJECTOR and LPOPT, respectively. We note that the later evaluation is of a special value illustrating the value and the potential of system PREDICTOR and technology of the kind. It assesses PREDICTOR's impact when it is used in practice for its intended purpose as a decision-making assistant. The intrinsic evaluation has its value in identifying potential future work directions and pitfalls in estimations. Overall, we will observe intrinsically that our estimates differ frequently in order of magnitude from the reality. Yet, extrinsic evaluation clearly states that PREDICTOR performs as a solid decision-making assistant for the purpose of improving rewriting tools when their performance depends on a decision when rewriting should take place versus not.

Benchmarks were gathered from two different sources. First, programs from the Fifth Answer Set Programming Competition (Calimeri *et al.* 2016) were used. Of the 26 programs in the competition, 13 were selected (these that system PROJECTOR, in its default settings, has preformed rewritings on). For each program, the **20** instances (originally selected for the competition) were used. One interesting thing to note about these encodings is that they are generally already well optimized. As such, performing projections often leads to an increase in grounding size. Second, benchmarks were gathered from an application called ASPCCG implementing a natural language parser (Lierler and Schüller 2012). This domain has been extensively studied in Buddenhagen and Lierler (2015) and was used to evaluate system PROJECTOR by Hippen and Lierler (2019). In that evaluation, the authors considered 3 encodings from ASPCCG: ENC1, ENC7, ENC19. We

https://doi.org/10.1017/S1471068423000078 Published online by Cambridge University Press

Table 1. *Key information for benchmark programs*

Program	**Keys**
Bottle filling	–
Hanoi tower	–
Incremental scheduling	`precedes/2[1]`, `importance/2[1]`, `job_device/2[1]`, `job_len/2[1]`, `deadline/2[1]`, `curr_job_start/2[1]`, `curr_on_instance/2[1]`, `instances/2[1]`
Knight tour with holes	–
Labyrinth	–
Minimal diagnosis	`obs_elabel/3[1,2]`
Nomystery	`at/2[1]`, *fuel*`/2[1]`, `goal/2[1]`
Permutation pattern matching	`t/2[1]`, `p/2[1]`
Ricochet robots	`amo/2[1]`, `d1/2[1]`, `dir/2[1]`
Solitaire	–
Stable marriage	`manAssignsScore/3[1,2]`, `womanAssignsScore/3[1,2]`
Valves location	`dem/3[1,2]`
Weighted sequence	`leafWeightCardinality/3[1]`
ASPCCG ENC1; ENC7; ENC19	`word_at/2[2]`, `category_tag_nofeatures/3[1]`, `category_tag/3[1]`, `adjacent/2[1]`

introduced changes to the encodings ENC1, ENC7, and ENC19 to make these in ASP-Core-2 standard Calimeri *et al.* (2020) compatible with the LPOPT system. We utilize the same **60** instances as in the mentioned evaluation of PROJECTOR. In our experiments, system PROJECTOR was provided with the key information for some root predicate arguments within several of the benchmarks. Non-default keys used for all benchmarks can be found in Table 1. The sign "-" within the table denotes benchmarks where no key information was provided by the user.

Table 2 details interesting features in the programs from both domains. The second column provides information about some features present in the programs. These features are abbreviated with the meanings as follows (abbreviation letters bolded): **n**on-tight program, **a**ggregates, **b**inary operation terms, **c**hoice rules, and **f**unction terms. The competition benchmarks also consisted of two encodings: a newer 2014 encoding and a 2013 encoding from the previous year. The third column specifies which encoding was used (in case the newer encoding consisted of no projections).

All tests were conducted on Ubuntu 18.04.3 with an Intel® Xeon® CPU E5-1620 v3 @ 3.50GHz and 32 GB of RAM. Furthermore, Python version 3.7.3 and PYCLINGO version 5.4.0 are used to run PREDICTOR. Grounding and solving was done by CLINGO version 5.4.0. For all benchmarks execution was limited to 5 min.

5.3.1 Intrinsic evaluation

Let S be the true grounding size of an instance of a program computed by GRINGO – that is, the number of rules in a ground program produced by GRINGO. Let S' be the grounding size predicted by PREDICTOR for the same instance. We define a notion of an *error factor* on a program instance as S'/S. The *average error factor* of a program/benchmark is the average of all error factors across the instances of a program. Table 3 shows the average

https://doi.org/10.1017/S1471068423000078 Published online by Cambridge University Press

Table 2. *Feature and version details for benchmark programs*

Program	**Features**	**2013**
Bottle filling	a,b	**Yes**
Hanoi tower	b	No
Incremental scheduling	a,b,c	No
Knight tour with holes	n,b	No
Labyrinth	n	No
Minimal diagnosis	n	No
Nomystery	a,b,c,f	No
Permutation pattern matching	c,b	No
Ricochet robots	n,a,b,c	No
Solitaire	a,b,c	No
Stable marriage	–	**Yes**
Valves location	n,a,c,f	No
Weighted sequence	n,c,b	**Yes**
ASPCCG ENC1; ENC7; ENC19	n,a,b,c,f	N/A

Table 3. *Average error factor for benchmark programs, with and without keys*

Program	**Average Error Factor**	**Average Error Factor (Keyless)**
Hanoi tower	–	1.5
Nomystery	1.5	1.5
Permutation pattern matching∗	**3.8**	5.0
Solitaire	–	4.3
Stable marriage	**3.7**	$7.5 * 10^{5}$
Bottle filling	–	$4.9 * 10^{9}$
Incremental scheduling∗	$1.1 * 10^{5}$	$1.1 * 10^{5}$
Labyrinth∗	–	$1.3 * 10^{1}$
Minimal diagnosis	$8.2 * 10^{3}$	$8.2 * 10^{3}$
Valves location∗	$\mathbf{1.3 * 10^{1}}$	$1.6 * 10^{1}$
ASPCCG ENC1	$\mathbf{2.9 * 10^{1}}$	$3.1 * 10^{1}$
ASPCCG ENC7	$\mathbf{1.3 * 10^{1}}$	$1.4 * 10^{1}$
ASPCCG ENC19	$2.2 * 10^{1}$	$2.2 * 10^{1}$
Knight tour with holes	–	$1.9 * 10^{-4}$
Ricochet robots	$2.0 * 10^{-1}$	$\mathbf{2.2 * 10^{-1}}$
Weighted sequence	$6.0 * 10^{-3}$	$\mathbf{1.1 * 10^{-2}}$

error factor using PRD-PROJECTOR for all programs [3]. The third column presents the case for programs when no key information is provided. Sign "–" indicates that for this

[3] The numbers presented for the ASPCCG ENC1, ENC7, ENC19 are due to the original encoding of these benchmarks non-compatible with the ASP-Core-2 standard and utilized in the experiments by Hippen and Lierler (2021).

https://doi.org/10.1017/S1471068423000078 Published online by Cambridge University Press

benchmark no key information was provided within the main encoding. The average error factor shown was rounded to make comparisons easier. An asterisk (∗) next to a benchmark name indicates that not all 20 instances of this benchmark were grounded within the allotted time limit. For instance, 19 instances of the *Incremental Scheduling* benchmark were successfully grounded, while the remaining instance timed out. For the benchmarks annotated by ∗ we only report the average error factor assuming the instances grounded successfully.

We partition the results into three groups using the average error factor. The partition is indicated by the horizontal lines on Table 3. First, there are five programs where the estimates computed by PREDICTOR are, on average, less than one order of magnitude over. Second, there are eight programs that are, on average, greater than one order of magnitude over. Finally, three programs are predicted to have lower grounding sizes than in reality.

We also note the impact that keys have on certain programs. We especially emphasize the difference in error between *Stable Marriage* with and without keys, where the average error factor is different by 5 orders of magnitude. The numbers in bold mark instances in which information on keys change the prediction.

It is obvious that the accuracy of system PREDICTOR could still use improvements. In many cases the accuracy is drastically erroneous. These results are not necessarily surprising. We identify five main reasons for observed data on PREDICTOR:

1. Insufficient data modeling is one weak point of PREDICTOR. Since we do not keep track what actual constants could be present in the ground extensions of a predicate, it is often the case that we overestimate argument size due to our inability to identify repetitive values.
2. Since we only identified keys for root predicate arguments, many keys were likely missed; automatic key detection is the direction of future work.
3. System PREDICTOR has limited support for such common language extensions as aggregates.
4. System PREDICTOR is vulnerable to what is known as *error propagation* (Ioannidis and Christodoulakis 1991).
5. While one might typically expect PREDICTOR to overestimate due to its limited capabilities in detecting repeated data, the underestimation on *Knight Tour with Holes*, *Ricochet Robots*, and *Weighted Sequence* programs is not surprising due to the fact that these programs are non-tight and utilize binary operations in terms.

5.3.2 Extrinsic evaluation

Here, we examine the *relative* accuracy of system PREDICTOR alongside PROJECTOR and LPOPT. In other words, we measure the quality of PREDICTOR by analyzing the impact it has on PROJECTOR and LPOPT performance. We recall that in all experiments we consider that PREDICTOR is provided information on keys as documented in Table 1.

Let S be the grounding size of an instance of a program, where grounding is produced by GRINGO. Let S' be the grounding size of the same instance in a modified (rewritten) version of the program. In this context, the modified version will either be the logic program outputted after using PROJECTOR/LPOPT or the logic program outputted after

https://doi.org/10.1017/S1471068423000078 Published online by Cambridge University Press

Table 4. *Average grounding size factors, and execution time factors for* PROJ *and* PRD-PROJ

Program	Grounding Size Factor		Execution Time Factor		Svd.	Svd.	Svd.
	PROJ	PRD-PROJ	PROJ	PRD-PROJ	**Svd.**	PROJ	PRD-PROJ
Hanoi tower	1.41	1.00	1.67	1.00	20	20	20
Inc. scheduling∗	1.14	1.12	1.06	1.10	13	13	14
Minimal diagnosis	1.06	1.00	1.04	1.00	20	20	20
Solitaire	1.41	1.00	1.32	0.99	19	19	19
Stable marriage	0.13	0.12	0.18	0.17	19	19	19
Bottle filling	1.36	1.36	1.44	1.43	20	20	20
Labyrinth	1.11	1.11	5.26	5.27	18	18	18
Perm. pattern match.∗ †	0.13	0.13	0.14	0.14	16	20	20
Valves location†	1.00	1.00	1.03	0.93	3	3	3
Weighted sequence†	1.00	1.00	3.05	1.59	19	16	17
ASPCCG ENC1	1.01	1.01	1.65	2.28	60	60	60
ASPCCG ENC7	0.90	1.00	1.57	2.20	60	60	60
ASPCCG ENC19	0.70	0.81	1.71	2.59	60	60	60
Knight tour with holes	0.80	0.90	0.50	2.45	1	1	1
Nomystery	0.62	1.00	1.23	1.00	7	8	7
Ricochet robots	0.91	1.00	0.85	1.00	20	20	20

using PRD-PROJECTOR/PRD-LPOPT. The *grounding size factor* of a program's instance is defined as S'/S. As such, a grounding size factor greater than 1 indicates that the modification increased the grounding size, whereas a value less than 1 indicates that the modification improved/decreased the grounding size. The *average grounding size factor* of a benchmark is the average of all grounding size factors across the instances of a benchmark. While we target improving the grounding size of a program, the ultimate goal is to improve the overall performance of ASP grounding/solving. Thus, we also compare the execution time of the programs, as that is ultimately what we want to reduce. Let S be the execution time of an answer set solver CLINGO (including grounding and solving) on an instance of a benchmark. Let S' be the execution time of CLINGO on the same instance in a modified version of the benchmark. The *execution time factor* of a program's instance is defined as S'/S. The *average execution time factor* of a benchmark is the average of all *execution time factors* across the instances of a benchmark.

Table 4 displays the average grounding size factor together with the average execution time factor for PROJECTOR and PRD-PROJECTOR on all benchmark programs. An asterisk (∗) following a program name indicates that not all 20 instances were grounded. In these cases, the average grounding size factor was only computed from instances where all 3 versions of the program (original, PROJECTOR, PRD-PROJECTOR) completed solving. The same concerns the computation of the average execution time factor. While we only consider instances in where all 3 version of the program completed grounding and then solving, we have included the exact number of instances grounded and solved by each version of the program, to show that the factors presented may be misleading.

https://doi.org/10.1017/S1471068423000078 Published online by Cambridge University Press

Table 5. *Average grounding size factors, and execution time factors for* LPOPT *and* PRD-LPOPT

	Grounding Size Factor		Execution Time Factor				
Program	LPOPT	PRD-LPOPT	LPOPT	PRD-LPOPT	**Svd.**	**Svd.** LPOPT	**Svd.** PRD-LPOPT
ASPCCG ENC1	0.92	0.89	0.88	0.87	60	60	60
ASPCCG ENC7	0.80	0.75	0.83	0.79	60	60	60
Hanoi tower	1.41	1.00	1.59	0.99	20	19	20
Minimal diagnosis	1.17	1.00	1.13	1.00	20	20	20
Bottle filling	1.00	0.28	0.98	0.39	20	20	20
Valves location	1.00	1.00	1.00	0.96	3	3	3
Solitaire∗	1.03	1.01	4.53	0.94	18	18	18
Knight tour with holes	3.36	2.18	1.28	0.90	1	1	1
Labyrinth	1.24	1.12	10.45	9.36	18	18	18
Weighted sequence†	1.07	1.04	1.13	2.11	19	20	20
Stable marriage†	1.01	1.01	1.02	1.02	19	19	19
Perm. pattern match.∗†	0.14	0.14	1.15	0.89	16	19	20
Inc. scheduling	1.78	2.30	1.01	1.24	13	13	13
ASPCCG ENC19	0.78	0.87	0.92	0.93	60	60	60
Nomystery	0.70	0.95	1.06	2.72	7	8	8
Ricochet robots	1.09	1.18	1.01	2.02	20	20	20

For example, consider program *Inc. Scheduling*, while PRD-PROJECTOR seems to have a slightly slower execution time than PROJECTOR alone, PRD-PROJECTOR managed to solve an additional instance, reflected by the decreased grounding time, therefore it would not be accurate to say the PROJECTOR outperformed PRD-PROJECTOR on that encoding. A dagger (†) following a program name indicates that there was a slight improvement for PRD-PROJECTOR, however this information was lost for the precision shown.

We partition the results into three sets, indicated by the horizontal lines on Table 4. The first set denotes programs in which PREDICTOR improved the grounding size factor of the program, the second set denotes programs in PREDICTOR did not have a noticeable effect on the grounding size factor, and the last set denotes programs in which PREDICTOR harmed the grounding size factor of the program as compared to the rewriting without predictions. We note that there are five programs in which PRD-PROJECTOR reduces the grounding size when compared to PROJECTOR, five programs in which PRD-PROJECTOR does not impact the grounding size, and six programs in which PRD-PROJECTOR increases the grounding size. By gray, highlight we mark the benchmarks where decrease in grounding size by means of using PREDICTOR resulted in the increase of solving time.

Table 5 displays the average grounding size factor together with the average execution time factor for LPOPT and PRD-LPOPT on all benchmark programs. It is data is organized in the same style as within Table 4 comparing PROJECTOR and PRD-PROJECTOR. We note that there are ten programs in which PRD-LPOPT reduces the grounding size when compared to LPOPT, two programs in which PRD-LPOPT does not impact the grounding size, and four programs in which PRD-LPOPT increases the grounding size.

https://doi.org/10.1017/S1471068423000078 Published online by Cambridge University Press

Overall, the results illustrate the validity of PREDICTOR approach. The system has especially positive impact within its integration with LPOPT. Also, the presented experimental data illustrates once more the importance of the development rewriting techniques and the possibility of their positive impact. Together with that decision support systems exemplified by PREDICTOR have to be designed and engineered to achieve the whole potential of ASP. We trust that system PREDICTOR is a solid step in that direction providing room for numerous improvements to account for nontrivial language features of ASP dialects.

6 Conclusions and future work

We introduced a method for predicting grounding size of answer set programs. We implement the described method in stand-alone system PREDICTOR that runs agnostic to any answer set grounder/solver pair. We expect this tool to become a foundation to decision support systems for rewriting/preprocessing tools in ASP. Indeed, using PREDICTOR as a decision support guide to rewriting system PROJECTOR and LPOPT improves their outcome overall. The same is observed for the case of the rewriting system called LPOPT. This proves the validity of the proposed approach, especially as further methods for improving estimation accuracy are explored in the future. As such system PREDICTOR is a unique tool unparalleled in earlier research ready for use within preprocessing frameworks in ASP. As discussed in the introduction: this work provides an important step towards achieving a goal of *truly* declarative answer set programming.

The section on intrinsic evaluation indicated a number of potential areas worth of improving estimations. It is one of the future work directions. Another one is utilizing PREDICTOR within other preprocessing tools of ASP. We trust that both efforts can be now undertaken as a community effort given the availability and transparency of PREDICTOR. Also, rather sophisticated techniques such as database-inspired optimizations, back-jumping, rewritings, binder splitting techniques are available in modern implementations of grounders (Gebser *et al.* 2011b; Calimeri *et al.* 2017). As of now these techniques are not accounted for when estimates are produced. Also at the moment, uniform distribution of values between the maximum and minimum in predicate arguments is assumed. Looking into different assumptions is also an interesting future direction.

Competing interests

The author(s) declare none.

References

Abiteboul, S., Hull, R. and Vianu, V. 1995. *Foundations of Databases*. Addison-Wesley.

Balduccini, M., Gelfond, M. and Nogueira, M. 2006. Answer set based design of knowledge systems. *Annals of Mathematics and Artificial Intelligence*, *47*, 1–2, 183–219.

Bichler, M. 2015. Optimizing non-ground answer set programs via rule decomposition. Bachelor Thesis, TU Wien.

Bichler, M., Morak, M. and Woltran, S. 2020. Lpopt: A rule optimization tool for answer set programming. *Fundamenta Informaticae*, *177*, 3–4, 275–296.

https://doi.org/10.1017/S1471068423000078 Published online by Cambridge University Press

Brewka, G., Eiter, T. and Truszczynski, M. 2011. Answer set programming at a glance. *Communications of the ACM*, *54*, 12, 92–103.

Buddenhagen, M. and Lierler, Y. 2015. Performance tuning in answer set programming. 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., vol. 9345. Lecture Notes in Computer Science. Springer, 186–198.

Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Maratea, M., Ricca, F. and Schaub, T. 2020. ASP-Core-2 input language format. *Theory and Practice of Logic Programming*, *20*, 2, 294–309.

Calimeri, F., Fusca, D., Perri, S. and Zangari, J. 2017. 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 Zangari, J. 2019. Optimizing answer set computation via heuristic-based decomposition. *Theory and Practice of Logic Programming*, *19*, 4, 603–628.

Dodaro, C., Galatà, G., Grioni, A., Maratea, M., Mochi, M. and Porro, I. 2021. An ASP-based solution to the chemotherapy treatment scheduling problem. *Theory and Practice of Logic Programming*, *21*, 6, 835–851.

Eiter, T., Fink, M., Tompits, H., Traxler, P. and Woltran, S. 2006. Replacements in non-ground answer-set programming. In *Proceedings, Tenth International Conference on Principles of Knowledge Representation and Reasoning, Lake District of the United Kingdom, June 2-5, 2006*, P. Doherty, J. Mylopoulos and C. A. Welty, Eds. AAAI Press, 340–351.

Eiter, T., Traxler, P. and Woltran, S. 2006. An implementation for recognizing rule replacements in non-ground answer-set programs. In *Logics in Artificial Intelligence, 10th European Conference, JELIA 2006, Liverpool, UK, September 13–15, 2006, Proceedings*, M. Fisher, W. van der Hoek, B. Konev, and A. Lisitsa, Eds., vol. 4160. Lecture Notes in Computer Science. Springer, 477–480.

Faber, W., Leone, N. and Perri, S. 2012. The intelligent grounder of DLV. In *Correct Reasoning* 2012. Springer, 247–264.

Fages, F. 1994. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, *1*, 51–60.

Gebser, M., Kaminski, R., Kaufmann, B., Lindauer, M., Ostrowski, M., Romero, J., Schaub, T. and Thiele, S. 2015. *Potassco User Guide*, Second edition. Institute for Informatics, University of Potsdam.

Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T. 2011a. Challenges in answer set solving. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays in Honor of Michael Gelfond*, M. Balduccini and T. Son, Eds., vol. 6565. Springer, 74–90.

Gebser, M., Kaminski, R., König, A. and Schaub, T. 2011b. Advances in gringo series 3. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer, 345–351.

Hippen, N. and Lierler, Y. 2019. Automatic program rewriting in non-ground answer set programs. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 19–36.

Hippen, N. and Lierler, Y. 2021. Estimating grounding sizes of logic programs under answer set semantics. In *Logics in Artificial Intelligence*, W. Faber, G. Friedrich, M. Gebser and M. Morak, Eds. Cham. Springer International Publishing, 346–361.

Hoos, H., Lindauer, M. T. and Schaub, T. 2014. claspfolio 2: Advances in algorithm selection for answer set programming. *TPLP*, *14*, 4–5, 569–585.

Ioannidis, Y. E. and Christodoulakis, S. 1991. On the propagation of errors in the size

https://doi.org/10.1017/S1471068423000078 Published online by Cambridge University Press

of join results. Technical report, University of Wisconsin-Madison Department of Computer Sciences.

Kaminski, R. and Schaub, T. 2022. On the foundations of grounding in answer set programming. *Theory and Practice of Logic Programming*, 1–60.

Lierler, Y. and Schüller, P. 2012. Parsing combinatory categorial grammar via planning in answer set programming. In *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, E. Erdem, J. Lee, Y. Lierler, and D. Pearce, Eds., vol. 7265. *Lecture Notes in Computer Science*. Springer, 436–453.

Liu, L., Truszczynski, M. and Lierler, Y. 2022. A machine learning system to improve the performance of ASP solving based on encoding selection. In *Logic Programming and Nonmonotonic Reasoning - 16th International Conference, LPNMR 2022, Genova, Italy, September 5-9, 2022, Proceedings*, G. Gottlob, D. Inclezan, and M. Maratea, Eds., vol. 13416. Lecture Notes in Computer Science. Springer, 415–428.

Maratea, M., Pulina, L. and Ricca, F. 2014. The multi-engine ASP solver ME-ASP: Progress report. *CoRR*, *abs/1405.0876*.

Mastria, E., Zangari, J., Perri, S. and Calimeri, F. 2020. A machine learning guided rewriting approach for ASP logic programs. In *Proceedings 36th International Conference on Logic Programming (Technical Communications), ICLP Technical Communications 2020, (Technical Communications) UNICAL, Rende (CS), Italy, 18-24th September 2020*, F. Ricca, A. Russo, S. Greco, N. Leone, A. Artikis, G. Friedrich, P. Fodor, A. Kimmig, F. A. Lisi, M. Maratea, A. Mileo, and F. Riguzzi, Eds., vol. 325. *EPTCS*, 261–267.

Ricca, F., Grasso, G., Alviano, M., Manna, M., Lio, V., Iiritano, S. and Leone, N. 2012. Team-building with answer set programming in the Gioia-Tauro seaport. *Theory and Practice of Logic Programming*, *12*, 3, 361–381.

Silberschatz, A., Korth, H. F. and Sudarshan, S. 1997. *Database System Concepts*, vol. 4. McGraw-Hill New York.

Ullman, J. D. 1988. *Principles of Database and Knowledge-Base Systems, Volume I*, vol. 14. Principles of Computer Science Series. Computer Science Press.