Estimating Grounding Sizes of Logic Programs under Answer Set Semantics

Nicholas Hippen and Yuliya Lierler

University of Nebraska at Omaha, Omaha NE 68182, USA
{nhippen,ylierler}@unomaha.edu

Abstract. Answer set programming (ASP) 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 a system PREDICTOR (and its algorithmic backend) for estimating the grounding size of programs, a metric that can influence a program's performance. We evaluate an impact of PREDICTOR when used as a guide for rewritings produced by the ASP rewriting tool PROJECTOR. The results demonstrate potential to this approach.

Keywords: Answer set programming · Language optimization.

1 Introduction

Answer set programming (ASP) [3] 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. ASP has been successfully used in scientific and industrial applications.

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 [13]. 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 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" [13]. Intelligent grounding techniques [10] utilized by grounders such as GRINGO [14] or IDLV [5] also keep such a reduction in mind. Intelligent grounding procedures analyze a given program to produce a smaller propositional program without altering the solutions. In addition, researchers looked into automatic program rewriting procedures. Systems such as SIMPLIFY [8,9], LPOPT [1,2], PROJECTOR [15] rewrite non-ground programs targeting the reduction of the grounding size. These systems are meant to be prepossessing tools agnostic to the later choice of ASP solving technology.

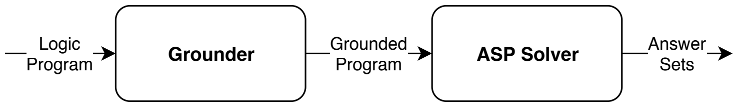

Fig. 1: Typical ASP system architecture

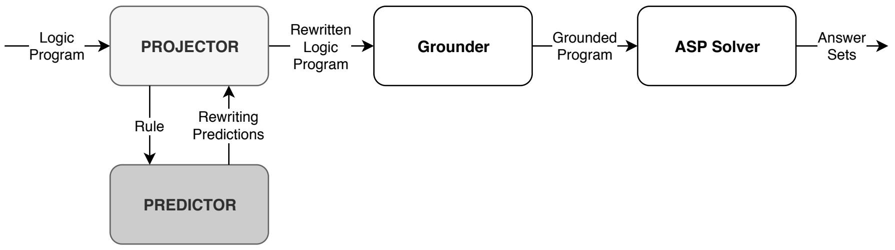

Fig. 2: An ASP system with PROJECTOR and PREDICTOR

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 is taken as a metric. Research has been performed to mitigate the negative impact of these rewritings. Mastria et al. [18] demonstrated a novel approach to guiding automatic rewriting techniques performed in IDLV using machine learning with a set of features built from structural properties and domain information. Calimeri et al. [6] illustrated truly successful application of a program rewriting technique stemming from LPOPT by incorporating its procedure inside the intelligent grounding algorithm of grounder IDLV. It was achieved by making a decision on whether to apply an LPOPT rewriting based on the current state of grounding. IDLV accurately estimated the impact of rewriting on grounding and based on this information decided 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 in [10] and (ii) techniques used in query optimization in relational databases (see, for instance, Chapter 13 in [19]). 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 tools PREDICTOR and PROJECTOR depicted in Figure 2. We illustrate the true success of this synergy by extensive experimental analysis. It is important 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 start by introducing the subject matter terminology. The key contribution of the work lays 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 arbitrary programs are given. We then describe the implementation details of system PREDICTOR. We conclude by experimental evaluation that includes incorporation of PREDICTOR within system PROJECTOR.

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 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 $\perp$, 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 set

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

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 in $\{\mathbb{H}(r)^i \mid r \in \Pi \text{ and } \mathbb{H}(r)^0 = p\}$; whereas $oc(\Pi)$ denotes the set of all object constants occurring in the head atoms of the rules in Π. For instance, consider a program, named Π_1:

$$p(1).\ p(2).\ r(3). \tag{3}$$

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

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

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

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

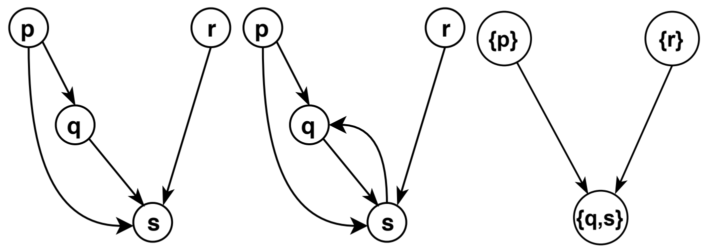

Fig. 3: Left: Graph G_{Π_2}; Center: Graph G_{Π_3}; Right: Graph $G^{sc}_{\Pi_3}$

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 $gr(\Pi)$. For instance, a 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 $\{\langle 1,1\rangle, \langle 2,1\rangle, \langle 3,1\rangle\}$ of tuples. 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* [11]. For instance, consider program Π_2 constructed from Π_1 by extending it with rules:

$$r(2).\ r(4). \tag{7}$$

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

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 system PREDICTOR (its algorithmic and software base), whose goal is to provide estimates for the size of an "intelligently" grounded program. PREDICTOR is based on the intelligent grounding procedures implemented by grounder DLV [10]. 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.

Argument size estimation Tight program case: The estimation formulas are based on predicting argument sizes. To understand these it is essential to talk about 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 *level rank* of a predicate. For example, p, q, r, s is one possible ordering for program Π_2. This ordering associates level 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 in relational databases, e.g., see Chapter 13 in [19]. These formulas keep track of information that helps us to guess what the 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 a process we introduce, 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 set $\{r \mid r \in \Pi,\ \mathbb{H}(r)^0 = p, \text{ and } \mathbb{H}(r)^i \text{ is a variable}\}$. 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:

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

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 level rank defined before (multiple level 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.

Now that we have estimates for minimum and maximum values, we estimate the size of the range of 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:

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

Recall, program $\varPi_2$. The operations required to compute the minimum estimate for argument $s[1]$ in $\varPi_2$ 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

$$\begin{aligned}
&min(\{max\big(\{0, \uparrow^{t\text{-}t}_{est}(s[1]) - \downarrow^{t\text{-}t}_{est}(s[1]) + 1\}\big), |oc(\varPi_2)|\})\\
&= min(\{max\big(\{0, 2-2+1\}\big), 4\}) = 1
\end{aligned}$$

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 a 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 a possible 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 milder the effect of overestimation we incorporate range estimates discussed before into the described computations.

For a tight program $\varPi$, let $p[i]$ be an argument in $\varPi$; R be the set

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

By $S^{t\text{-}t}_{est}(p[i])$ we denote an estimate of the argument size $p[i]$ in tight program $\varPi$. This estimate is computed as follows:

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

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.

The following illustrates the computation of the argument size estimates for argument $s[2]$ in program $\mathit{\Pi}_2$, given that $range^{t\text{-}t}_{est}(s[2]) = 2$ and $oc(s[2]) = \emptyset$:

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

Arbitrary (nontight) program case: To process arbitrary programs (tight and non-tight), we must manage to resolve the circular dependencies such as present in sample program $\mathit{\Pi}_3$ defined in the section on preliminaries. We borrow and simplify a concept of the component graph from [10]. The *component graph* of a program $\mathit{\Pi}$ is an acyclic directed graph $G^{sc}_{\mathit{\Pi}} = \langle N, E\rangle$ such that N is the set of strongly connected components in the dependency graph $G_{\mathit{\Pi}}$ of $\mathit{\Pi}$ and E contains the edge (P, Q) if there is an edge (p, q) in $G_{\mathit{\Pi}}$ 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 $\mathit{\Pi}_3$. For a program $\mathit{\Pi}$, 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 level rank* of each predicate that belongs to a node. For example, $\{p\}, \{r\}, \{q, s\}$ is one possible topological sorting of $G^{sc}_{\mathit{\Pi}_3}$. This ordering associates the following strong level ranks $1, 2, 3, 3$ with predicates p, r, q, s, respectively.

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

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

We call this set a *module*. A rule r in module $\mathcal{P}_C$ 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. For instance, the modules in program $\mathit{\Pi}_3$ contain

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

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

In the sequel we consider components whose module contains an exit rule. For a component C and its module $\mathcal{P}_C$, we construct a partition $M_1, ..., M_n$ $(n \geq 1)$ in the following way: Every exit rule of $\mathcal{P}_C$ is a member of M_1. A recursive rule r in $\mathcal{P}_C$ 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). The component partition of node $\{q, s\}$ in $G^{sc}_{\Pi_3}$ follows:

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

For a component partition $M_1, \ldots, M_k, \ldots, M_n$, by $M^{p[i]}_k$ 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 $M^{p[i]}_{1\ldots k}$ we denote the union $\bigcup_{j=1}^{k} M^{p[i]}_j$. For instance, for program Π_3 and its argument $q[1]$:

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

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 relied 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 components 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 $1 \le j \le n$.

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:

$$\downarrow_{est}(p[i]) = \downarrow^{gr}_{est}(p[i], n)$$

$$\downarrow^{gr}_{est}(p[i], j) = min(oc(p[i]) \cup \{\downarrow^{rule}_{est}(p[i], j, r) \mid r \in M^{p[i]}_{1\ldots j}\})$$

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

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

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 level ranking of program's predicates. Then, by $rank(p)$ we refer to the corresponding strong level 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 $\mathit{\Pi}$. 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:

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

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

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

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

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. 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:

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

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, the primary key is manually determined.) It is possible that some predicates do not have primary keys specified. To handle such predicates, we define $key(p)$ to mean the following:

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

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 subsystem of answer set solving toolkit CLINGO [12]. 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.

4 Experimental Analysis

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 system PRD-PROJECTOR– a tool resulting from system PROJECTOR enhanced by PREDICTOR– is conducted. In particular, we investigate the utility of system PREDICTOR by integrating it as a decision support mechanism into the ASP rewriting tool PROJECTOR. This integration is illustrated in Figure 2. Each time system PROJECTOR accounts a rule to which its rewriting is applicable, it performs the rewriting. System PRD-PROJECTOR performs the rewriting of PROJECTOR only if PREDICTOR predicts the reduction in grounding size upon the rewriting. We measure the quality of PREDICTOR by analyzing the impact it has on rewritings by PROJECTOR. We note that the extrinsic evaluation is of a special value illustrating the usefulness and the potential of system PREDICTOR. 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

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

that our estimates differ frequently in order of magnitude from the reality. Yet, extrinsic evaluation clearly states that PREDICTOR performs as an excellent 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 [7] were used. Of the 26 programs in the competition, 13 were selected (these that system PROJECTOR 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 [17]. This domain has been extensively studied in [4] and was used to evaluate system PROJECTOR in [15]. In that evaluation, the authors considered 3 encodings from ASPCCG: ENC1, ENC7, ENC19. We utilize the same encodings and instances as in the evaluation of PROJECTOR. 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 minutes.

Intrinsic Evaluation Let S be the true grounding size of an instance in a program computed by GRINGO. Let S' be the grounding size predicted by PREDICTOR of 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 1 shows the average error factor for all programs. We note that in our tests, keys were manually identified only for root predicate arguments. 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 instances timed out. For the $*$ benchmarks 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 1. 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. 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. (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* [16]. (5) While one might

Program	**Avg. Error Factor**
Hanoi Tower	1.5
Nomystery	1.5
Perm. Pattern Match.$*$	3.8
Solitaire	4.3
Stable Marriage	3.7
Bottle Filling	4.9×10^9
Inc. Scheduling$*$	1.1×10^5
Labyrinth$*$	1.3×10^1
Minimal Diagnosis	8.2×10^3
Valves Location	$\mathbf{1.3 \times 10^1}$
ASPCCG ENC1	$\mathbf{2.9 \times 10^1}$
ASPCCG ENC7	$\mathbf{1.3 \times 10^1}$
ASPCCG ENC19	2.2×10^1
Knight Tour with Holes	1.9×10^{-4}
Ricochet Robots	$\mathbf{2.0 \times 10^{-1}}$
Weighted Sequence	$\mathbf{6.0 \times 10^{-3}}$

Table 1: Average error factor for benchmark programs

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.

Extrinsic Evaluation Here, we examine the *relative* accuracy of system PREDICTOR alongside PROJECTOR. In other words, we measure the quality of PREDICTOR by analyzing the impact it has on PROJECTOR performance.

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 or the logic program outputted after using PRD-PROJECTOR. 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. Table 2 (left) displays the average grounding size 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 grounding. 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 2 (left). We note that there are eight programs in which PRD-PROJECTOR reduces the grounding size noticeably when compared to PROJECTOR, five programs in which PRD-PROJECTOR does not impact the grounding size noticeably, and three programs in which PRD-PROJECTOR increases the grounding size noticeably.

Program	PROJ	PRD-PROJ
Hanoi Tower	1.41	1.00
Inc. Scheduling∗	1.14	1.12
Minimal Diagnosis	1.06	1.00
Solitaire	1.41	1.00
Stable Marriage	0.13	0.12
ASPCCG ENC1	0.63	0.49
ASPCCG ENC7	1.40	1.24
ASPCCG ENC19	1.58	1.04
Bottle Filling	1.36	1.36
Labyrinth∗	1.11	1.11
Perm. Pattern Match.∗ †	0.13	0.13
Valves Location†	1.00	1.00
Weighted Sequence†	1.00	1.00
Knight Tour with Holes	0.80	0.90
Nomystery	0.62	1.00
Ricochet Robots	0.91	1.00

Program	**Svd.**	PROJ	PRD-PROJ
Hanoi Tower	20	1.67	1.00
Inc. Scheduling	13	1.06	1.10
Minimal Diagnosis	20	1.04	1.00
Solitaire	19	1.32	0.99
Stable Marriage	19	0.18	0.17
ASPCCG ENC1	54	0.57	0.52
ASPCCG ENC7	57	1.37	1.28
ASPCCG ENC19	59	1.93	1.16
Bottle Filling	20	1.44	1.43
Labyrinth	16	5.26	5.27
Perm. Pattern Match.	16	0.14	0.14
Valves Location	3	1.03	0.93
Weighted Sequence	16	3.05	1.59
Knight Tour with Holes	1	0.50	2.45
Nomystery	7	1.23	1.00
Ricochet Robots	20	0.85	1.00

Table 2: Left: Average grounding size factors; Right: Average execution time factors

While we target improving the grounding size of a program, it is useful to 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 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 2 (right) shows the *average execution time factor* of programs rewritten with PROJECTOR and PRD-PROJECTOR. Overall, the results illustrate the validity of PREDICTOR approach.

5 Conclusions

We introduced a method for predicting grounding size of answer set programs. To the best of our knowledge this is *the only* approach for the stated purpose. 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 improves the PROJECTOR's outcome overall. 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 such as SIMPLIFY or LPOPT in a similar manner as we illustrate its use here within the system PRD-PROJECTOR.

Acknowledgments 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.

References

1. Bichler, M.: Optimizing non-ground answer set programs via rule decomposition. (2015), Bachelor Thesis, TU Wien
2. Bichler, M., Morak, M., Woltran, S.: lpopt: A rule optimization tool for answer set programming. In: Proceedings of International Symposium on Logic-Based Program Synthesis and Transformation (2016)
3. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM **54**(12), 92–103 (2011)
4. Buddenhagen, M., Lierler, Y.: Performance tuning in answer set programming. In: Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR) (2015)
5. Calimeri, F., Fusca, D., Perri, S., Zangari, J.: I-dlv: The new intelligent grounder of dlv. Intelligenza Artificiale **11**(1), 5–20 (2017)
6. Calimeri, F., Fusca, D., Perri, S., Zangari, J.: Optimizing answer set computation via heuristic-based decomposition. In: International Symposium on Practical Aspects of Declarative Languages. pp. 135–151. Springer (2018)
7. Calimeri, F., Gebser, M., Maratea, M., Ricca, F.: Design and results of the fifth answer set programming competition. Artificial Intelligence **231**, 151 – 181 (2016). https://doi.org/https://doi.org/10.1016/j.artint.2015.09.008, http://www.sciencedirect.com/science/article/pii/S0004370215001447
8. Eiter, T., Fink, M., Tompits, H., Traxler, P., Woltran, S.: Replacements in non-ground answer-set programming. In: Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR) (2006)
9. Eiter, T., Traxler, P., Woltran, S.: An implementation for recognizing rule replacements in non-ground answer-set programs. In: Proceedings of European Conference On Logics In Artificial Intelligence (JELIA) (2006)
10. Faber, W., Leone, N., Perri, S.: The intelligent grounder of dlv. In: Correct Reasoning, pp. 247–264. Springer (2012)
11. Fages, F.: Consistency of Clark's completion and existence of stable models. Journal of Methods of Logic in Computer Science **1**, 51–60 (1994)
12. Gebser, M., Kaminski, R., Kaufmann, B., Lindauer, M., Ostrowski, M., Romero, J., Schaub, T., Thiele, S.: Potassco user guide. Institute for Informatics, University of Potsdam, second edition edition (2015)
13. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Challenges in answer set solving. In: Balduccini, M., Son, T. (eds.) Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays in Honor of Michael Gelfond, vol. 6565, pp. 74–90. Springer (2011)
14. Gebser, M., Schaub, T., Thiele, S.: Gringo: A new grounder for answer set programming. In: Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning. pp. 266–271 (2007)
15. Hippen, N., Lierler, Y.: Automatic program rewriting in non-ground answer set programs. In: International Symposium on Practical Aspects of Declarative Languages. pp. 19–36. Springer (2019)
16. Ioannidis, Y.E., Christodoulakis, S.: On the propagation of errors in the size of join results. Tech. rep., University of Wisconsin-Madison Department of Computer Sciences (1991)
17. Lierler, Y., Schueller, P.: Parsing combinatory categorial grammar with answer set programming: Preliminary report (2011), http://www.cs.utexas.edu/users/ai-lab/pub-view.php?PubID=127116

18. Mastria, E., Zangari, J., Perri, S., Calimeri, F.: A machine learning guided rewriting approach for asp logic programs. arXiv preprint arXiv:2009.10252 (2020)
19. Silberschatz, A., Korth, H.F., Sudarshan, S., et al.: Database system concepts, vol. 4. McGraw-Hill New York (1997)