12-2019

Grounding Size Predictions for Answer Set Programs

Nicholas Hippen
University of Nebraska at Omaha

Grounding Size Predictions for Answer Set Programs

A Thesis

Presented to the

Department of Computer Science

and the

Faculty of the Graduate College

University of Nebraska

In Partial Fulfillment
of the Requirements for the Degree

Master of Science

University of Nebraska at Omaha

by

Nicholas Hippen

December 2019

Supervisory Committee:

Dr. Yuliya Lierler

Dr. Abhishek Parakh

Dr. Victor Winter

Grounding Size Predictions for Answer Set Programs

Nicholas Hippen, MS

University of Nebraska at Omaha, 2019

Advisor: Dr. Yuliya Lierler

Answer set programming is a declarative programming paradigm geared towards solving difficult combinatorial search problems. Logic programs under answer set semantics can typically be written in many different ways while still encoding the same problem. These different versions of the program may result in diverse performances. Unfortunately, it is not always easy to identify which version of the program performs the best, requiring expert knowledge on both answer set processing and the problem domain. More so, the best version to use may even vary depending on the problem instance. One measure that has been shown to correlate with performance is the programs grounding size, a measure of the number of ground rules in the grounded program (Gebser et al. 2011). Computing a grounded program is an expensive task by itself, thus computing multiple ground programs to assess their sizes to distinguish between these programs is unrealistic. In this research, we present a new system called PREDICTOR to estimate the grounding size of programs without the need to actually ground/instantiate these rules. We utilize a simplified form of the grounding algorithms implemented by answer set programming grounder DLV while borrowing techniques from join-order size estimations in relational databases. The PREDICTOR system can be used independent of the chosen answer set programming grounder and solver system. We assess the accuracy of the predictions produced by PREDICTOR, while also evaluating its impact when used as a guide for rewritings produced by the automated answer set programming rewriting system called PROJECTOR. In particular, system PREDICTOR helps to boost the performance of PROJECTOR.

This thesis is dedicated to my family for their love and support throughout my education.

Acknowledgements

Coming into the senior year of my Bachelor's degree and start of my Master's degree at the University of Nebraska at Omaha (UNO), I was curious but uncertain of what to expect from research. Thankfully, I had the pleasure of working with and being advised by Yuliya Lierler, who made my experience very enjoyable. I want to thank her for the endless support she provided me and for giving me the opportunity to conduct research.

I would also like to thank the other members of my thesis committee, Abhishek Parakh and Victor Winter, for providing helpful feedback and serving on my committee.

I am grateful to have worked with Mirek Truszczynski as well as his students at the University of Kentucky: Daniel Houston, Liu Liu, Michael Dingess, and Shelby Stocker, with whom I had many discussions.

I also appreciate the assistance from Roland Kaminski and Parvathi Chundi in answer various questions.

Finally, I would like to thank all past and present members of the Natural Language Processing and Knowledge Representation lab (NLPKR) at UNO that I have worked with: Da Shen, Gang Ling, John Hare, Brian Hodges, Craig Olson, and Justin Robbins.

Table of Contents

List of Figures

List of Tables

1 Introduction

Answer set programming (ASP) (Brewka et al. 2011) is a declarative programming paradigm geared towards knowledge representation and solving difficult combinatorial search problems. Unlike problem solutions utilizing procedural programming, answer set programs are defined declaratively, leaving only the task of modeling the application as a set of logic rules for the programmer. These logic rules define a problem instance to be solved. An ASP system is then capable of producing all solutions / answer sets that are supported by these rules through automated reasoning. Essentially, ASP systems provide a generic search platform to the developer.

Typical ASP system architecture consists of a two step process, depicted in Figure 1 (Lierler 2017). The first step transforms a non-ground logic program into a semantically equivalent program without variables (such a program is called a ground program). This step expands the number of logic rules in the program, often significantly. The number of logic rules in a grounded program is referred to as the grounding size of a program. The grounding step is the focus of our attention in this thesis. We introduce this step at greater detail in Section 2.

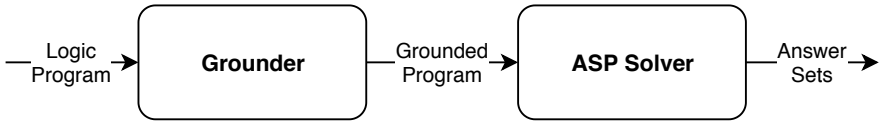

Figure 1: Typical ASP system architecture

The second step of ASP systems is to solve the grounded logic program and produce the answer sets, also referred to as stable models. The search algorithms implemented in ASP solvers are closely related to those in the field of SAT solving (Lierler 2017).

The task of producing an ASP solution to a problem is reduced to modeling the problem's search space and its constraints. As a result, developing a solution to some problems can be much easier than the same task using an imperative algorithm tailored to a domain. In fact, it is often the case that problems requiring complex search can be solved more effectively

using SAT than procedural algorithms designed specifically for the problem instance thanks to the number of optimizations in SAT solvers (Rossi et al. 2008). Due to the relation between SAT and ASP solving procedures it is reasonable to believe the same observation is applicable in ASP.

However, the intuitive ASP encodings are not always the most optimal. As in imperative programming paradigms, ASP programs often require careful design and expert knowledge in order to achieve performant results (Bichler et al. 2016). One way to mitigate this issue is to introduce automated rewriting techniques that alleviate the burden of optimization from the programmer. Here, we focus on rewriting techniques performed on non-ground logic programs (ASP logic programs prior to being input to a grounder) (Bichler 2015, Bichler et al. 2016, Eiter, Fink, Tompits, Traxler & Woltran 2006, Eiter, Traxler & Woltran 2006, Hippen & Lierler 2019). While many rewriting techniques of this kind exist, not all of them guarantee that the rewritten program is solved faster. Grounding size has been shown to be predictive of a programs performance, enabling it to be used as an "optimality" metric (Gebser et al. 2011). Unfortunately, the grounding step is usually expensive and accounts for a reasonable chunk of an ASP system's runtime. Thus, obtaining the grounding size by grounding in order to elicit light on the potential performance of a given encoding is unrealistic.

To solve this issue we have implemented a new system, called PREDICTOR, designed to estimate the grounding size of a non-ground logic program. This system utilizes statistics gathered from a basic parsing of the program in order to extrapolate information about the grounding size. To achieve this, we utilize a simplified form of the grounding algorithms implemented by answer set programming grounder DLV (Faber et al. 2012) while taking inspiration from join-order size estimations in relational databases (Silberschatz et al. 1997). System PREDICTOR is developed to be used independent of the grounding and solving system chosen.

Thesis Outline In Section 2, we start by presenting necessary notation for understanding

ASP logic programs, the grounding procedures used in ASP systems, and additional information to understand the implementation of PREDICTOR. We continue by describing the ASP optimization tool, PROJECTOR (Hippen & Lierler 2019), which we later utilize for evaluation of system PREDICTOR. In Section 3 we discuss motivating work and present the problem statement. In Section 4 we present the implementation details of system PREDICTOR. In Section 5 we describe how PREDICTOR is integrated into PROJECTOR. In Section 6 we provide both an intrinsic evaluation of the accuracy of PREDICTOR and an extrinsic evaluation of PREDICTOR when used as a guide for rewritings produced by PROJECTOR. Finally, in Section 7 we summarize our findings and describe potential future work.

2 Preliminaries

2.1 Basic Terminology

We first consider the vocabulary necessary to understanding the syntax and semantics of the ASP formalism. 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. We say atom of this form is *defined by* predicate symbol p. For an atom a and an index $1 \leq i \leq k$, by a^i we denote the term t_i. A *term* is either an object constant or a variable.

Example 2.1 An atom $p(1, X, Y)$ is such that

- 1 is an object constant
- symbols X, Y are variables (here we use the standard convention for identifiers in logic programming where they start with a capital letter to denote variables), and
- p is a predicate symbol of arity 3.

A *rule* of a logic program is 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

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

as the *body* of the rule. We refer to atoms a and their negations $not\ a$ as literals. To literals $a_1, ..., a_m$ 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 and by $\mathbb{B}^-(r)$ the set of negative literals in the body of r. We also use $\mathbb{B}(r)$ to denote $\mathbb{B}^+(r) \cup \mathbb{B}^-(r)$. We say that r is positive if $\mathbb{B}^-(r) = \emptyset$. We call a rule with an empty body ($\mathbb{B}(r) = \emptyset$) a *fact* while a rule whose head is $\perp$ we call a *constraint*. We can obtain the set of variables present in an atom a by $vars(a)$. We can obtain the set of variables present in a rule r by $vars(r)$. A rule r is *safe* if each variable in r appears in $\mathbb{B}^+(r)$.

Example 2.2 Let a be the atom $q(A, B)$, then $vars(a) = \{A, B\}$.

Example 2.3 Let r be the rule

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

Then $vars(r) = \{A, B\}$.

Example 2.4 Rule (2) is safe while both

$$p(A).$$
$$p(A) \leftarrow not\ q(A).$$

are unsafe rules.

A *logic program* is a finite set of safe rules. We call programs containing variables as *non-ground* ASP programs. For an ASP program Π, we use $p[i]$ to identify predicate

arguments, where p is a predicate and i is a valid argument index for the predicate.

Example 2.5 Consider the following non-ground ASP program Π_1:

$$
\begin{aligned}
& p(1).\ p(2). \\
& r(3). \\
& q(X, 1) \leftarrow p(X).
\end{aligned} \tag{3}
$$

Predicate argument $p[1]$ is valid for Π_1 while $p[2]$ is not because the argument index 2 is not valid for p.

By $oc(p[i])$ we denote the set of object constants occurring in the head atom defined by predicate p at argument index i for all rules. We denote the cardinality of $oc(p[i])$ by $|oc(p[i])|$. For an ASP program Π, we extend this notation so that $oc(\Pi)$ denotes the set of object constants occurring in head atoms of all rules in Π.

Example 2.6 Let us consider the set of object constants and their cardinalities for program Π_1. We can see that:

$$
\begin{aligned}
& oc(p[1]) = \{1, 2\} \\
& |oc(p[1])| = 2 \\
& oc(r[1]) = \{3\} \\
& oc(q[1]) = \emptyset \\
& oc(q[2]) = \{1\} \\
& oc(\Pi_1) = \{1, 2, 3\} \\
& |oc(\Pi_1)| = 3
\end{aligned} \tag{4}
$$

2.2 Grounding Algorithms of DLV

Typical ASP technology requires that an ASP program is grounded before it is solved. The grounding process involves instantiating variables in the program with all object constants of the program, producing a program without variables.

Example 2.7 Recall Π_1 and its set of object constants given in (4). Grounding program Π_1 will produce the following ground program, $gr(\Pi_1)$:

$$
\begin{aligned}
& p(1).\ p(2). \\
& r(3). \\
& q(1,1) \leftarrow p(1). \\
& q(2,1) \leftarrow p(2). \\
& q(3,1) \leftarrow p(3).
\end{aligned} \tag{5}
$$

For some ASP program Π, by $|gr(\Pi)|$ we denote the *grounding size* of $gr(\Pi)$, where we understand the grounding size of a program as the number of rules present in the grounded program. For some rule r in Π, by $|gr_r(\Pi)|$ we denote the *grounding size* of r with respect to program Π, where we understand the grounding size of a rule r as the number of rules present in the ground program generated by r (all rules resulting from the instantiation of r).

Example 2.8 Let r be rule (3). Then,

$$
\begin{aligned}
|gr(\Pi_1)| &= 6 \\
|gr_r(\Pi_1)| &= 3
\end{aligned}
$$

It is vital to ASP systems that these ground programs are computed efficiently. Grounding procedures utilize "intelligent restrictions" that can be observed about the rules that usually decrease, often drastically, the grounding size (Lierler et al. 2016). As such, these *intelligently grounded* programs are a "subset" of the ground programs discussed before,

such as $gr(\Pi_1)$, while still having the same answer sets. Such a program is called an *image*. Grounding procedures are usually not straightforward.

Example 2.9 The following program, $igr(\Pi_1)$, is an image of $gr(\Pi_1)$:

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

The grounding size, which we denote by $|igr(\Pi_1)|$, is 5. Similarly, let r be rule (3). The grounding size of r, denoted by $|igr_r(\Pi_1)|$, is 2.

Notice how in this intelligent grounding, rule (5) is no longer present. Intuitively, this rule defined an impossible case, as $p(3)$ could never be true in Π_1. The rule can be eliminated from the grounding while still obtaining the same answer sets. Some systems capable of performing this task include LPARSE (Syrjänen 2000), GRINGO (Gebser et al. 2007), and IDLV (a newer version of DLV) (Calimeri et al. 2017). For this thesis, we are mainly looking at the intelligent grounding procedure implemented by DLV (Faber et al. 2012).

The *ground extensions* of a predicate within an intelligently grounded program $igr(\Pi)$ are the set of terms associated with the predicate in the program.

Example 2.10 In $igr(\Pi_1)$ from Example 2.9, the ground extensions of predicate p is the set of tuples

$$\{\langle 1\rangle, \langle 2\rangle\},$$

while the ground extensions of predicate q is the set of tuples

$$\{\langle 1,1\rangle, \langle 2,1\rangle\}.$$

For a predicate argument $p[i]$ and an intelligently grounding program $igr(\Pi)$, by $V(p[i])$

we denote the *argument size* of $p[i]$ to be the number of distinct object constants present in the ground extensions of p in $igr(\Pi)$ for the corresponding argument position i.

Example 2.11 The predicates in $igr(\Pi_1)$ from Example 2.9 have the following argument sizes:

$$V(p[1]) = 2$$
$$V(r[1]) = 1$$
$$V(q[1]) = 2$$
$$V(q[2]) = 1$$

For a rule r and a variable X in r, by $args(r, X)$ we denote the set of predicate arguments constructed as follows:

$$\{p[i] \mid X \text{ is an argument of some predicate } p \text{ at argument index } i \text{ in } \mathbb{B}^+(r)\}$$

Example 2.12 Let r be rule (2). Then,

$$args(r, A) = \{q[1], r[2]\}$$
$$args(r, B) = \{q[2]\}$$

2.3 Graphs

In this subsection we introduce several graph concepts important to this thesis. A *topological sort* of a directed acyclic graph $G = \langle N, E \rangle$ is an ordering of its nodes such that for every edge $(u, v) \in E$, u is placed before v in the ordering. It is possible that there are multiple topological sorts for any given graph.

Example 2.13 The graph given in Figure 2 has 3 possible topological sorts.

$$p, q, r, s$$

$$p, r, q, s$$

$$r, p, q, s$$

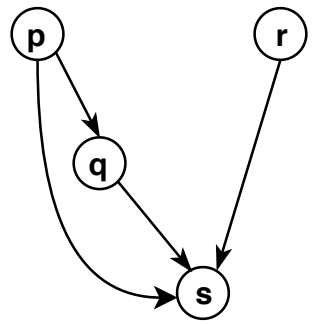

Figure 2: Example acyclic graph for topological sorting

Given a directed graph $G = \langle N, E \rangle$, a set of nodes $N' \subseteq N$ is said to be *strongly connected* if there exists a path between all nodes in N'. A *strongly connected component* is a maximal strongly connected set of nodes such that no additional nodes can be added to the set without making the set no longer strongly connected. Note that in an acyclic graph, all strongly connected components will contain exactly one node.

Example 2.14 The strongly connected components in the graph in Figure 3 are:

$$\{p\}, \{r\}, \{q, s\}$$

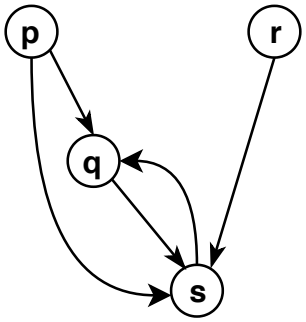

Figure 3: Example graph with cycles

2.4 System PROJECTOR

System PROJECTOR (Hippen & Lierler 2019) is a program designed to rewrite non-ground ASP programs. The goal of PROJECTOR is to reduce the grounding size of a program automatically. Figure 2.4 displays the intended use of PROJECTOR within a typical ASP solving infrastructure. PROJECTOR takes as input a non-ground ASP logic program and outputs a rewritten program. One can see that all rewriting occurs before any grounding is done. As such, PROJECTOR is agnostic to both the grounding and solving systems so long as the ASP dialect, ASP-Core-2 (Calimeri et al. 2012), supported by PROJECTOR is compatible with that of the chosen grounder and solver pair.

Figure 4: Typical ASP system architecture extended with PROJECTOR

Overall, the task of PROJECTOR is to divide a rule into multiple smaller rules such that each rule has fewer variables than the original. The technique utilized by PROJECTOR is inspired by projection rewriting used in SQL query optimization in relational databases (Faber et al. 1999). More specifically, each rule r is analyzed and a set V of variables is identified for projection. Set V can contain any combination of variables present in the body of r so long as they do not occur in the head of r. In order to remove these variables from the original rule, all literals containing variables in V are moved into a new rule, called the

projection rule. The projection rule defines a fresh predicate with respect to the program, whose terms correspond to the variables present in the projection rule that are not in V. The literals that are removed in the original rule are replaced with the new literal defined by the fresh predicate introduced in the projection rule. This revised version of the original rule is referred to as the *replacement rule*.

Example 2.15 Let us consider the following rule r:

$$p(K) \leftarrow q(K, P), r(K, L), s(L).$$

Because variables P, L are present only in the body, we can compute all possible sets (excluding the empty set) V to be

$$\{\{P, L\}, \{P\}, \{L\}\}$$

The projection of $V = \{P\}$ follows:

$$q'(K) \leftarrow q(K, P).$$
$$p(K) \leftarrow q'(K), r(K, L), s(L).$$

The projection of $V = \{L\}$ follows:

$$rs'(K) \leftarrow r(K, L), s(L).$$
$$p(K) \leftarrow q(K, P), rs'(K).$$

Note that there is no projection for $\{P, L\}$, as the resulting projection rule coincides with the original rule.

For more detailed information on how to derive the results for the projections shown in Example 2.15 as well as details on PROJECTOR and its performance, we refer the reader

to (Hippen & Lierler 2019).

2.4.1 Key Issues of PROJECTOR

The algorithms used to compute intelligently grounded programs are expensive. Additionally, programs may have grounding sizes that are too large to deal with in memory, leaving grounding as either a major bottleneck or roadblock (Gebser et al. 2011). Selecting the best rewritings prior to or during grounding may alleviate some of the grounding bottleneck.

Many efforts have been put forth to perform automatic rewrites on non-ground ASP programs (Bichler 2015, Bichler et al. 2016, Eiter, Fink, Tompits, Traxler & Woltran 2006, Eiter, Traxler & Woltran 2006, Hippen & Lierler 2019), yet not all techniques guarantee that the rewritten program runs faster. Gebser et al. (2011) has provided a set of guidelines that can be used to help tune non-ground ASP programs. One such guideline suggests to watch out for grounding size, as reducing the grounding size of a program often leads to faster solve times. We take this guideline as the key to predicting the quality of rewritings, so that if a rewriting system produces a program whose grounding is smaller than that of an original program, then we consider the performance of a rewriting system as satisfactory.

As mentioned before, the main goal of PROJECTOR is to reduce the grounding size of a program automatically. Unfortunately, the rewritings performed by PROJECTOR do not guarantee a reduction in grounding size. In addition to this, there are cases where there are multiple rewritings possible for a single rule (because there are multiple candidates for V) and choosing one rewriting could eliminate the possibility of performing others. As such, in order to maximize the performance of a program, it is important to identify the best rewritings to perform, if any. It was identified that one of the major caveats of PROJECTOR is that the heuristics implemented may often not identify projections that result in a smaller grounding size. Additionally, the heuristics used in PROJECTOR do not support the ability to perform no rewrite at all. If a projection rewrite is possible it will always perform one. We address how these issues are solved in Section 5.

3 Motivating Work and Problem Statement

System LPOPT (Bichler 2015, Bichler et al. 2016) is an ASP program pre-processing tool that rewrites rules through *tree-decomposition*. Like PROJECTOR, system LPOPT may divide a rule into multiple smaller rules with the guarantee that each rule has fewer variables than the original. Unlike PROJECTOR, however, LPOPT derives its rewritings by converting a given rule into a tree and computing the tree-decompositions via the general-purpose library, *htd*[1]. An algorithm is then used to convert the decomposition back into multiple ASP rules to replace the original rule. Nonetheless, like PROJECTOR, the rewritings produced by LPOPT do not guarantee that the grounding size will be reduced.

Grounder IDLV has implemented the tree-decomposition based rewriting techniques introduced by LPOPT as part of its default optimizations (Calimeri et al. 2018). However, unlike LPOPT, grounder IDLV utilizes heuristics tailored towards its grounding procedures. In order to achieve this, they compute grounding size estimations of the rules produced in each rewriting. These estimations are used in deciding which rewriting decomposition, if any, to use. They found that this grounding-size heuristic-based approach on average improves the running time over both IDLV (without this optimization) and LPOPT combined in pipeline with IDLV. Indeed, IDLV is capable of predicting the grounding size of individual rules, however it is only capable of performing these predictions immediately before grounding the rule it is predicting. As such, it is not possible to predict the grounding size of an arbitrary rule prior to starting the grounding process. Naturally, it is also not possible to predict the grounding size of the entire program. These predictions are also not portable; they are tightly-coupled with the grounder IDLV. Thus, this cannot be used with other grounding systems as-is. The success of their approach led us to investigating the possibility of developing a stand alone tool that we call PREDICTOR whose functionality is to produce estimates for the grounding size of a given program or rule in some context without performing grounding itself.

[1] https://github.com/mabseher/htd

4 PREDICTOR System Implementation

System PREDICTOR is based on the grounding procedures implemented by grounder DLV (Faber et al. 2012). The primary difference is that, instead of building the ground instances of each rule in the program, PREDICTOR builds statistics about the predicates and their arguments. This system is capable of producing estimates of the grounding size for most non-ground ASP programs.

4.1 Estimation Formulas

We begin this subsection by introducing the fundamental concepts for understanding the order to compute argument size estimations. We then introduce estimation formulas in two parts. First, we describe a methodology for computing estimations for a class of logic programs called *tight* programs. We then extend these formulas to work for arbitrary programs.

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. We say that a predicate q *depends on* some predicate p if there exists a path from p to q. A program Π is *tight* if G_Π is acyclic, otherwise the program is *non-tight*.

Example 4.1 Recall program Π_1 from Example 2.5. Program Π_2 is the program Π_1 extended with a few additional rules. Program Π_2 is shown below:

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

$$r(2).r(3).r(4).$$

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

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

Program Π_3 is the program Π_2 extended with the rule:

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

Figure 5 shows the dependency graphs for Π_2 (left) and Π_3 (right). Notice how G_{Π_2} is acyclic while G_{Π_3} is not.

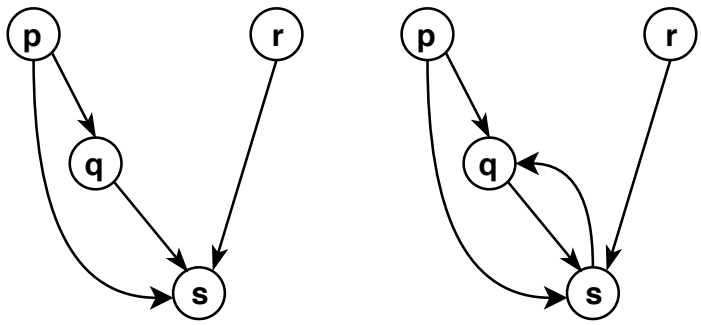

Figure 5: Left: The dependency graph G_{Π_2}; Right: The dependency graph G_{Π_3}.

The dependency graph serves well as a visualization for how to compute argument size estimates based on dependencies. In G_{Π_2}, it is easy to see that before we compute argument size estimates for predicate arguments of predicate s, we compute the estimates for p, q, and r. Similarly before we compute estimates for q we compute the estimates for p. One can obtain the order to compute all estimates of the predicates of Π_2 by performing a topological sort on G_{Π_2}. However, when looking at G_{Π_3} there is an issue. In order to compute the estimates for s, we compute the estimates for p, q, and r. However, in order to compute the estimates for q, we compute the estimates for p and s. Because s depends on q and q depends on s, we have a circular dependency between q and s. Initially, we only consider the case of tight programs, however we discuss how we handle this issue in Section 4.1.2.

It is convenient to build on the dependency graph to use predicate arguments instead of predicates. The *argument dependency graph* of a program Π is a directed graph $G^a_\Pi = \langle N, E \rangle$ such that N is the set of valid predicate arguments in Π and E contains the edge $(p[i], q[i'])$ if there is a rule r in Π in which $p[i]$ contains a variable in $\mathbb{B}^+(r)$

and $q[i']$ contains that same variable in the head of r. We call those predicates arguments with no incoming edges *root predicate arguments*.

Example 4.2 Recall programs Π_2 and Π_3 from Example 4.1. Figure 6 shows the argument dependency graph for Π_2 (left) and Π_3 (right).

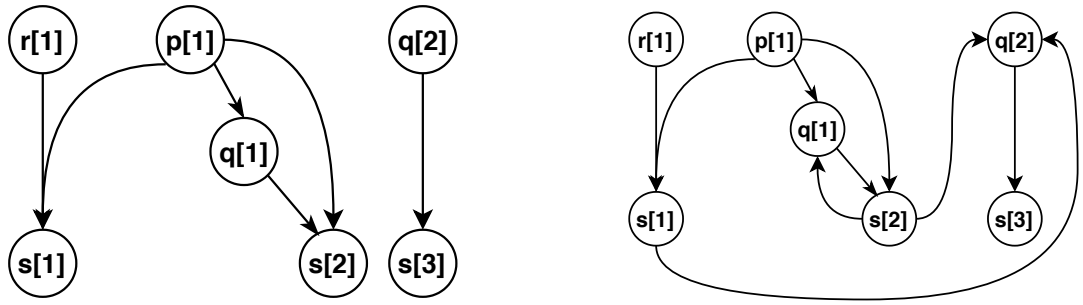

Figure 6: Left: The argument dependency graph $G^a_{\Pi_2}$; Right: The argument dependency graph $G^a_{\Pi_3}$.

In program Π_2, predicate arguments $r[1]$, $p[1]$, and $q[2]$ are root predicate arguments. In program Π_3, predicate arguments $r[1]$ and $p[1]$ are root predicate arguments.

4.1.1 Tight Programs

We now introduce some intermediate formulas for constraining our estimates based on the data. These intermediate formulas are inspired by query optimization techniques in relational databases, e.g., see Chapter 13 in (Silberschatz et al. 1997).

Data Distribution It is useful to keep track of some information that helps us to guess what the actual values may be in the grounded program without storing all values. Let $p[i]$ be a predicate argument. We now define a useful concept of numeric predicate arguments.

If $p[i]$ is a root predicate argument, we consider $p[i]$ as *numeric* if all values in $oc(p[i])$ are numeric. If $p[i]$ is not a root predicate argument, then $p[i]$ is numeric if there exists no non-numeric $q[i']$ such that there is a path from $q[i']$ to $p[i]$ in the argument dependency graph.

Example 4.3 All predicate arguments in Π_2 from Example 4.1 are numeric. However, if we

append the following fact to the program:

$$r(a).$$

, then $r[1]$ and $s[1]$ become non-numeric.

We introduce a methodology for tracking the range of values for predicate arguments that are numeric. To provide intuitions for the processes, consider the following intelligent grounding of Π_2 from Example 4.1:

$$p(1).p(2).$$

$$r(2).r(3).r(4).$$

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

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

$$s(2,1,1) \leftarrow r(2), p(2), p(1), q(1,1). \tag{9}$$

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

Note that the intelligent grounding of rule (7) produces rules (9), (10), while variable X from rule (7) is only ever replaced with the object constant 2. Intuitively, this is due to the intersection $oc(p[1]) \cap oc(r[1]) = \{2\}$. We attempt to model this restriction by considering what minimum and maximum values are possible for each predicate argument in the intelligently grounded program. We then use these values to define an "upper restriction" to the argument size for each predicate argument.

We begin with intuitions behind definitions of minimum and maximum estimations for some predicate argument $p[i]$. Consider the case of maximum estimates. If $p[i]$ is a root predicate argument, we simply find the maximum value of $oc(p[i])$. Otherwise, let us consider some rule with a head atom defined by p containing a variable X at argument position i. This rule gives rise to multiple ground rules in the processes of ground instantiation. Among

these ground rules, we want to identify what the *maximum value* appears at $p[i]$ in the heads. To do so, we find the maximum estimates for predicate arguments in the positive body of the rule that contain X. Based on typical intelligent grounding procedures, we know that X will never be instantiated by a value greater than the minimum of those estimates. Here, we found the maximum estimate relative to a single rule. Intuitively, the maximum estimate of $p[i]$ with respect to the entire program is the maximum of values computed in this manner for each rule and $oc(p[i])$. In the case of minimum estimates, we mirror the ideas behind maximum estimates. We now proceed to the formalization of this estimation procedure.

For a predicate argument $p[i]$ in some program Π, we define the minimum and maximum estimates for tight programs, $min_{est}^{tight}(p[i])$ and $max_{est}^{tight}(p[i])$, respectively, as follows:

- If $p[i]$ is a root predicate argument in the argument dependency graph of the program, we simply use the smallest object constant present in the heads containing an atom defined by p for argument position i, i.e.

$$min_{est}^{tight}(p[i]) = min\big(oc(p[i])\big)$$
$$max_{est}^{tight}(p[i]) = max\big(oc(p[i])\big)$$

- Otherwise, let R represent the set of rules $r \in \Pi$ such that $\mathbb{H}(r)$ is defined by p and $\mathbb{H}(r)^i$ is a variable. Then,

$$min_{est}^{tight}(p[i]) = min\Big(oc(p[i]) \cup \Big\{max\big(\{min_{est}^{tight}(p'[i']) \mid p'[i'] \in args(r, \mathbb{H}(r)^i)\}\big) \mid r \in R\Big\}\Big)$$

$$max_{est}^{tight}(p[i]) = max\Big(oc(p[i]) \cup \Big\{min\big(\{max_{est}^{tight}(p'[i']) \mid p'[i'] \in args(r, \mathbb{H}(r)^i)\}\big) \mid r \in R\Big\}\Big)$$

We note that these recursive definitions are well defined as we are in the case of tight

programs so that the estimates for "body" predicate arguments present recursively in the definition are always computed prior to "head" predicate arguments.

Now that we have estimates for minimum and maximum values, we estimate the size of the range of values. We understand the *range* of a predicate argument to be the number of values we anticipate to see in the predicate argument within the 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 predicate argument is greater than its maximum estimate. Intuitively, we understand that this indicates no ground rule will contain the predicate argument in their heads. The number of values between the minimum and maximum estimates may also be greater than the number of object constants in the program. Naturally, in this case it makes sense to restrict the range to the number of object constants in the program. We compute the range, $range_{est}^{tight}(p[i])$ as follows:

$$range_{est}^{tight}(p[i]) = min\Big(\Big\{max\big(\{0, max_{est}^{tight}(p[i]) - min_{est}^{tight}(p[i]) + 1\}\big), |oc(\Pi)|\Big\}\Big)$$

Example 4.4 The following shows the operations needed to compute the maximum estimate for predicate argument $s[1]$ in program Π_2 from Example 4.1:

$$\begin{aligned}
&max_{est}^{tight}(s[1]) = max\Big(oc(s[1]) \cup \Big\{min\big(\{max_{est}^{tight}(r[1]), max_{est}^{tight}(p[1])\}\big)\Big\}\Big)\\
&max_{est}^{tight}(r[1]) = max\big(oc(r[1])\big) = 4\\
&max_{est}^{tight}(p[1]) = max\big(oc(p[1])\big) = 2\\
&max_{est}^{tight}(s[1]) = max\Big(\emptyset \cup \Big\{min\big(\{4, 2\}\big)\Big\}\Big) = max\Big(\Big\{2\Big\}\Big) = 2
\end{aligned}$$

Without showing the intermediate operations needed, we note that $min_{est}^{tight}(s[1]) = 2$. Using

this, we can compute the range estimate for $s[1]$:

$$range_{est}^{tight}(s[1]) = min\Big(\Big\{max\big(\{0, max_{est}^{tight}(s[1]) - min_{est}^{tight}(s[1]) + 1\}\big), |oc(\Pi_2)|\Big\}\Big)$$
$$range_{est}^{tight}(s[1]) = min\Big(\Big\{max\big(\{0, 2 - 2 + 1\}\big), |oc(\Pi_2)|\Big\}\Big) = min\Big(\Big\{1, 4\Big\}\Big) = 1$$

Note that these equations assume that considered predicate arguments are numeric. If predicate argument $p[i]$ is not numeric, we assume the following:

$$range_{est}^{tight}(p[i]) = |oc(\Pi)|$$

Argument Size Estimates We begin with an informal definition of argument size estimations for some predicate argument $p[i]$. If $p[i]$ is a root predicate argument, we simply estimate that the argument size is $|oc(p[i])|$. Otherwise, let us consider some rule with a head atom defined by p containing some variable X at argument position i. We want to find the *number of values* X could be replaced with in an intelligent grounding. To do so, we find the argument size estimate for predicate arguments in the positive body of the rule that contain X. Based on typical intelligent grounding procedures, we know that X can never be more values than the minimum of those argument size estimations. Here, we have only found the argument size estimate relative to a single rule, but the argument size estimate of $p[i]$ with respect to the entire program is the sum of the number of values computed in this manner for each rule, in addition to $|oc(p[i])|$. It is easy to see that the sum over all rules may heavily overestimate the argument size. We use our range estimation discussed before to restrict the estimation. We now proceed to the formalization of the argument size estimation procedure.

We note that the general pattern of this formula is very similar to that of the minimum and maximum estimation formulas. Let $p[i]$ be a predicate argument in some program Π. We define a argument size estimate for tight programs, $V_{est}^{tight}(p[i])$, as follows:

- If $p[i]$ is a root predicate argument in the argument dependency graph of the program,

we simply count the number of distinct object constants present in the heads containing an atom defined by p for argument position i, i.e.

$$V_{est}^{tight}(p[i]) = |oc(p[i])|$$

- Otherwise, let R represent the set of rules $r \in \Pi$ such that $\mathbb{H}(r)$ is defined by p and $\mathbb{H}(r)^i$ is a variable. Then,

$$V_{est}^{tight}(p[i]) = min\Big(\Big\{|oc(p[i])| + \sum_{r \in R} min\big(\{V_{est}^{tight}(p'[i']) \mid p'[i'] \in args(r, \mathbb{H}(r)^i)\}\big), \\ range_{est}^{tight}(p[i])\Big\}\Big)$$

Example 4.5 The following shows the operations needed to compute the argument size estimates for predicate argument $s[2]$ in program Π_2 from Example 4.1, given that $range_{est}^{tight}(s[2]) = 2$:

$$V_{est}^{tight}(s[2]) = min\Big(\Big\{|oc(s[2])| + min\big(\{V_{est}(p[1]), V_{est}(q[1])\}\big), range_{est}^{tight}(s[2])\Big\}\Big)$$
$$V_{est}^{tight}(p[1]) = |oc(p[1])| = 2$$
$$V_{est}^{tight}(q[1]) = min\Big(\Big\{|oc(q[1])| + min\big(\{V_{est}(p[1])\}\big), range_{est}^{tight}(q[1])\Big\}\Big)$$
$$V_{est}^{tight}(q[1]) = min\Big(\Big\{0 + min(\{2\}), 2\Big\}\Big) = 2$$
$$V_{est}^{tight}(s[2]) = min\Big(\Big\{0 + min\big(\{2, 2\}\big), 2\Big\}\Big) = 2$$

4.1.2 Arbitrary Programs

To be able to process arbitrary programs (i.e. both tight and non-tight programs), we must manage to resolve the circular dependency such as the one present in Π_3. We introduce a new graph to visualize how to resolve this issue. This graph is a simplified version of the component graph introduced in (Faber et al. 2012), altered for the necessary functions of PREDICTOR.

The *simple 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 and E contains the edge (P, Q) if there is an edge (p, q) in G_{Π} where $p \in P$ and $q \in Q$. For tight programs, the simplified component graph will be effectively identical to its dependency graph, with the only difference being that the predicate nodes are replaced with predicate sets that only contain a single corresponding predicate each. Let p be a predicate in some component C of G^{sc}_{Π}. We will refer to C as the component of p. Figure 7 shows the simple component graphs for Π_2 (left) and Π_3 (right). Notice how both $G^{sc}_{\Pi_2}$ and $G^{sc}_{\Pi_3}$ are acyclic.

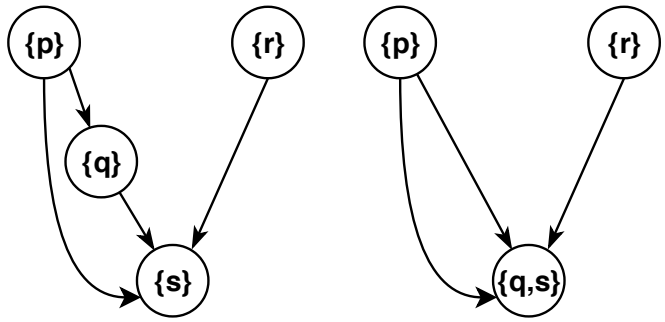

Figure 7: Left: The simple component graph $G^{sc}_{\Pi_2}$; Right: The simple component graph graph $G^{sc}_{\Pi_3}$.

Like with the dependency graph, we can use the simple component graph as a visualization for how to compute argument size estimates. We say that a predicate q *strongly depends on* some predicate p if there exists a path from P to Q where $p \in P$ and $q \in Q$. While tight programs have the same dependencies as strong dependencies, non-tight programs lose those dependencies between predicates that are grouped within the same component. It is important to note that this dependency loss makes PREDICTOR less suited towards non-tight programs. With tight programs, we compute estimates while only concerning ourselves with the order of estimating predicates. For non-tight programs, we consider both the order in which we evaluate predicates as well as the order in which we evaluate rules associated with components that consist of several predicates.

Let C be some node in G^{sc}_{Π}. We call a *module* of C, denoted by $\mathcal{P}_C$, as the set of rules whose head contains an atom defined by some predicate in C. A rule $r \in \mathcal{P}_C$ is a *recursive*

rule if there exists an atom in the positive body of r that defines some predicate $p \in C$; otherwise r is an *exit rule*. We say a predicate p is a *recursive predicate* if p occurs in the head of some recursive rule; otherwise it is an *exit predicate*. For tight programs, all rules are exit rules and all predicates are exit predicates. Note that it is possible to have modules with only recursive rules.

Example 4.6 The modules in program Π_3 from Example 4.1 are:

$$\mathcal{P}_{\{p\}} = \{p(1). \quad p(2).\}$$
$$\mathcal{P}_{\{r\}} = \{r(2). \quad r(3). \quad r(4).\}$$
$$\mathcal{P}_{\{q,s\}} = \{q(X,1) \leftarrow p(X). \quad s(X,Y,Z) \leftarrow r(X), p(X), p(Y), q(Y,Z).$$
$$q(Y,X) \leftarrow s(X,Y,Z).\}$$

The only recursive rules are:

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

Therefore, the recursive predicates are s, q while the exit predicates are p, r.

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: Let r be a rule in $\mathcal{P}_C$. If r is an exit rule, then r is in M_1. Otherwise, r is in M_k ($k > 1$) if for every predicate $p \in C$ occurring in $\mathbb{B}^+(r)$, there is a rule in $M_1 \cup ... \cup M_{k-1}$ such that its head atom is defined by p and there exists a predicate q occurring in $\mathbb{B}^+(r)$ such that there exists some rule in M_{k-1} where its head atom is defined by q. We refer to the unique partition created in this manner as the *component partition* of C. We call elements of a component partition *groups* (the component partition is undefined for components whose module does not contain an exit rule).

Example 4.7 The component partition of $\{q, s\}$ in Π_3 from Example 4.1 follows:

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

Let $p[i]$ be a predicate argument. We refer to the subset of rules in some group of a component partition by $M_k^{p[i]}$ $(k \geq 1)$ when it is the set of rules $r \in M_k$ such that $\mathbb{H}(r)$ is defined by p and $\mathbb{H}(r)^i$ is a variable. By $M_{1...k}^{p[i]}$ we denote the union $M_1^{p[i]} \cup ... \cup M_k^{p[i]}$.

Example 4.8 In program Π_3, for predicate 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 revisit range and argument size estimation formulas for tight programs and extend them for arbitrary programs. One may observe that these formulas are more complex than their respective tight versions, yet they perform similar operations at their core. Intuitively, formulas for tight programs relied on predicate argument ordering given by acyclic structure of a program's dependency graph. Here in addition to some order provided by the dependency graph we also rely on the order given to us by the component partition corresponding to a given program.

Data Distribution Let $p[i]$ be a numeric predicate argument in some program Π. Let C be the component of p, where the module of C contains an exit rule. Let n be the cardinality of the component partition of C, and j be an integer such that $1 \leq j \leq n$. We define the minimum estimation formula, $min_{est}(p[i])$, as follows:

$$min_{est}(p[i]) = min_{est}^{group}(p[i], n)$$

where

$$min_{est}^{group}(p[i], j) = \begin{cases} min_{est}^{tight}(p[i]), & \text{if } p \text{ is an exit predicate} \\ min\big(oc(p[i]) \cup \big\{min_{est}^{rule}(p[i], j, r) \mid r \in M_{1...j}^{p[i]}\big\}\big), & \text{otherwise} \end{cases}$$

where

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

where

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

The intermediate functions: $min_{est}^{group}(p[i], j)$, $min_{est}^{rule}(p[i], j, r)$, $min_{est}^{split}(p[i], p'[i'], j)$ provide a way to "shrink" the number of rules being considered for a given predicate argument. This is done through the "counter" argument, j, in order to avoid looping infinitely. We note the strong similarity between the combined definitions of $min_{est}^{group}(p[i], j, r)$ and $min_{est}^{rule}(p[i], j, r)$ compared to the corresponding tight formula $min_{est}^{tight}(p[i])$. Formula $min_{est}^{split}(p[i], p'[i'], j)$ serves two purposes. If the predicate p' is in the same component as p, we decrement the "counter" argument j. Otherwise, we simply use the minimum estimate for $p'[i']$ that is due to the computation relevant to another component. Note that we assume ordering on components provided by the simple component graph.

Like with tight programs, the maximum estimation formula is extremely similar to the minimum estimation formula. Let $p[i]$ be a numeric predicate argument in some program Π. Let C be the component of p, where the module of C contains an exit rule. Let n be the cardinality of the component partition of C, and j be an integer such that $1 \leq j \leq n$. We

define the maximum estimation formula, $max_{est}(p[i])$, as follows:

$$max_{est}(p[i]) = max_{est}^{group}(p[i], n)$$

where

$$max_{est}^{group}(p[i], j) = \begin{cases} max_{est}^{tight}(p[i]), & \text{if } p \text{ is an exit predicate} \\ max\big(oc(p[i]) \cup \big\{max_{est}^{rule}(p[i], j, r) \mid r \in M_{1\ldots j}^{p[i]}\big\}\big), & \text{otherwise} \end{cases}$$

where

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

where

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

Example 4.9 Let us name rules (6), (7), (8) as rules r_1, r_2, and r_3 respectively. The following shows the operations needed to compute the maximum estimate for predicate argument $q[1]$ in program Π_3 from Example 4.1. Recall the modules and component partition computed in Examples 4.6 and 4.7, respectively. We note that $oc(q[1]) = oc(s[2]) = \emptyset$ and $max_{est}^{tight}(p[1]) = 2$. Then,

$$\begin{aligned}
&max_{est}(q[1]) = max_{est}^{group}(q[1], 3) \\
&max_{est}^{group}(q[1], 3) = max\big(oc(q[1]) \cup \big\{max_{est}^{rule}(q[1], 3, r_1), max_{est}^{rule}(q[1], 3, r_3)\big\}\big) \\
&max_{est}^{rule}(q[1], 3, r_1) = min\big(\big\{max_{est}^{split}(q[1], p[1], 3)\big\}\big) \\
&\qquad\qquad = max_{est}(p[1]) = max_{est}^{group}(p[1], 1) = max_{est}^{tight}(p[1]) = 2 \\
&max_{est}^{rule}(q[1], 3, r_3) = min\big(\big\{max_{est}^{split}(q[1], s[2], 3)\big\}\big)
\end{aligned}$$

$$= max_{est}^{group}(s[2], 2)$$

$$= max\big(oc(s[2]) \cup \big\{max_{est}^{rule}(s[2], 2, r_2)\big\}\big)$$

$$max_{est}^{rule}(s[2], 2, r_2) = min\big(\big\{max_{est}^{split}(s[2], p[1], 2), max_{est}^{split}(s[2], q[1], 2)\big\}\big)$$

$$max_{est}^{split}(s[2], p[1], 2) = max_{est}(p[1]) = 2$$

$$max_{est}^{split}(s[2], q[1], 2) = max_{est}^{group}(q[1], 1)$$

$$= max\big(oc(q[1]) \cup \big\{max_{est}^{rule}(q[1], 1, r_1)\big\}\big)$$

$$max_{est}^{rule}(q[1], 1, r_1) = min\big(\big\{max_{est}^{split}(q[1], p[1], 0)\big\}\big)$$

$$= max_{est}(p[1]) = 2$$

$$max_{est}^{split}(s[2], q[1], 2) = max\big(\emptyset \cup \big\{2\big\}\big) = 2$$

$$max_{est}^{rule}(s[2], 2, r_2) = min\big(\big\{2, 2\big\}\big) = 2$$

$$max_{est}^{rule}(q[1], 3, r_3) = max\big(\emptyset \cup \big\{2\big\}\big) = 2$$

$$max_{est}^{group}(q[1], 3) = max(\emptyset \cup \{2, 2\}) = 2$$

$$max_{est}(q[1]) = 2$$

We compute the range estimate for arbitrary programs in the same manner as we compute the range estimates of tight programs. We replace the min and max formulas for tight programs with their corresponding arbitrary formulas:

$$range_{est}(p[i]) = min\Big(\Big\{max\big(\{0, max_{est}(p[i]) - min_{est}(p[i]) + 1\}\big), |oc(\Pi)|\Big\}\Big)$$

We also apply the same assumption for non-numeric arguments. Let $p[i]$ be non-numeric. Then,

$$range_{est}(p[i]) = |oc(\Pi)|$$

Furthermore, if the module of C does not contain an exit rule, then

$$range_{est}(p[i]) = 0$$

Example 4.10 The following shows the operations needed to compute the range estimate for predicate argument $q[1]$ in program Π_3 from Example 4.1. Recall from Example 4.9 that $max_{est}(q[1]) = 2$. We note that $min_{est}(q[1]) = 1$ and $|oc(\Pi_3)| = 4$. We compute the range estimate for $q[1]$:

$$range_{est}(q[1]) = min\Big(\Big\{max\big(\{0, max_{est}(q[1]) - min_{est}(q[1]) + 1\}\big), |oc(\Pi_3)|\Big\}\Big)$$
$$range_{est}(q[1]) = min\Big(\Big\{max\big(\{0, 2 - 1 + 1\}\big), |oc(\Pi_3)|\Big\}\Big) = min\Big(\Big\{2, 4\Big\}\Big) = 2$$

Argument Size Estimates Let $p[i]$ be a predicate argument in some program Π. Let C be the component of p, where the module of C contains an exit rule, and let n be the cardinality of the component partition of C. We define the formula for finding the argument size estimates, $V_{est}(p[i])$, as follows:

$$V_{est}(p[i]) = V_{est}^{group}(p[i], n)$$

where

$$V_{est}^{group}(p[i], j) = \begin{cases} V_{est}^{tight}(p[i]), & \text{if } p[i] \text{ is an exit predicate} \\ min\big(\{|oc(p[i])| + \sum_{r \in M_{1\ldots j}^{p[i]}} V_{est}^{rule}(p[i], j, r), & \\ \qquad range_{est}(p[i])\}\big), & \text{otherwise} \end{cases}$$

where

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

where

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

Like with the minimum and maximum estimation formulas, the intermediate functions: $V_{est}^{group}(p[i], j)$, $V_{est}^{rule}(p[i], j, r)$, $V_{est}^{split}(p[i], p'[i'], j)$ provide a way to "shrink" the number of rules being considered for a given predicate argument.

Furthermore, if the module of C does not contain an exit rule, then

$$V_{est}(p[i]) = 0$$

Example 4.11 Let us again recall rules (6), (7), (8) as rules r_1, r_2, and r_3 respectively. The following shows the operations needed to compute the argument size estimate for predicate argument $q[1]$ in program Π_3. We note that $range_{est}(s[2]) = 2$, $|oc(q[1])| = |oc(s[2])| = 0$, $|oc(\Pi_3)| = 4$, and $V_{est}^{tight}(p[1]) = 2$. Then,

$$V_{est}(q[1]) = V_{est}^{group}(q[1], 3)$$

$$V_{est}^{group}(q[1], 3) = min\big(\{|oc(q[1])| + V_{est}^{rule}(q[1], 3, r_1) + V_{est}^{rule}(q[1], 3, r_3), range_{est}(q[1]), |oc(\Pi_3)|\}\big)$$

$$\begin{aligned} V_{est}^{rule}(q[1], 3, r_1) &= min\big(\{V_{est}^{split}(q[1], p[1], 3)\}\big) \\ &= V_{est}(p[1]) = V_{est}^{group}(p[1], 1) = V_{est}^{tight}(p[1]) = 2 \end{aligned}$$

$$\begin{aligned} V_{est}^{rule}(q[1], 3, r_3) &= min\big(\{V_{est}^{split}(q[1], s[2], 3)\}\big) \\ &= V_{est}^{group}(s[2], 2) \\ &= min\big(\{|oc(s[2])| + V_{est}^{rule}(s[2], 2, r_2), range_{est}(s[2]), |oc(\Pi_3)|\}\big) \end{aligned}$$

$$V_{est}^{rule}(s[2], 2, r_2) = min\big(\{V_{est}^{split}(s[2], p[1], 2), V_{est}^{split}(s[2], q[1], 2)\}\big)$$

$$V_{est}^{rule}(s[2], p[1], 2) = V_{est}(p[1]) = 2$$

$$V_{est}^{split}(s[2], q[1], 2) = V_{est}^{group}(q[1], 1)$$

$$= min\big(\{|oc(q[1])| + V_{est}^{rule}(q[1], 1, r_1), range_{est}(q[1]), |oc(\Pi_3)|\}\big)$$

$$V_{est}^{rule}(q[1], 1, r_1) = min\big(\{V_{est}^{split}(q[1], p[1], 0)\}\big)$$

$$= V_{est}(p[1]) = 2$$

$$V_{est}^{split}(s[2], q[1], 2) = min\big(\{0 + 2, 2, 4\}\big) = 2$$

$$V_{est}^{rule}(s[2], 2, r_2) = min\big(\{2, 2\}\big) = 2$$

$$V_{est}^{rule}(q[1], 3, r_3) = min\big(\{0 + 2, 2, 4\}\big) = 2$$

$$V_{est}^{group}(q[1], 3) = min\big(\{0 + 2 + 2, 2, 4\}\big) = 2$$

Here we can also see the impact that tracking the range of values can have on argument size estimates. Had the object constants in Π_3 been considered non-numeric, we would have instead estimated that the argument size of $q[1]$ is 4.

Keys We now borrow the concept of keys from relational databases. For some predicate p, we refer to any set of predicate arguments of p that can uniquely identify all ground extensions of p as a *superkey* of p.

Example 4.12 Let the following be the ground extensions of p:

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

It is easy to see that both $\{p[1], p[2]\}$ and $\{p[1], p[2], p[3]\}$ are superkeys of p, while $\{p[1]\}$ is not a superkey.

We say a superkey K of p is a *candidate key* of p if there is no other superkey K' of p such that $K' \subset K$. In other words, a candidate key is a minimal superkey. From Example 4.12, only $\{p[1], p[2]\}$ is a candidate key. A *primary key* of p is a single chosen

candidate key. A predicate may have at most one primary key. For the purposes of this thesis, the primary key is manually determined. We discuss how the user specifies keys in Section 4.3. 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 a predicate argument $p[i]$ a *key predicate argument* if it is in $key(p)$.

Let r be a rule. By $kvars(r)$ we denote the set of all variables that occur only in key predicate arguments in rule r.

Example 4.13 Let r be the rule:

$$\perp \leftarrow p(X, Y, V), q(V), r(X). \tag{11}$$

and let the keys be:

$$keys(p) = \{p[1], p[2]\}$$
$$keys(q) = \{q[1]\}$$
$$keys(r) = \{r[1]\}$$

Then,

$$kvars(r) = \{X, Y\}.$$

Rule and Program Estimates We can now compute the estimated grounding size of rules.

Let Π be a logic program and let r be a rule in Π. The estimated grounding size of a rule, $igr_{est}(r)$, is the following:

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

Example 4.14 Let us refer to rule (11) as r. Recall its associated keys from Example 4.13. Given that $V_{est}(p[1]) = 3$, $V_{est}(p[2]) = 3$, and $V_{est}(r[1]) = 2$, we can compute the grounding size of the rule in the following way:

$$\begin{aligned} igr_{est}(r) &= min\big(\{V_{est}(p[1]), V_{est}(r[1])\}\big) * min\big(\{V_{est}(p[2])\}\big) \\ &= min\big(\{3, 2\}\big) * min\big(\{3\}\big) \\ &= 2 * 3 = 6 \end{aligned}$$

Naturally, we can also compute the estimated grounding size of some logic program Π in the following way:

$$\sum_{r \in \Pi} igr_{est}(r)$$

4.2 Language Extensions

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.

4.2.1 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 semi-colon 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 4.15 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 4.16 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.

4.2.2 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

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

where $n \geq 0$ and $e_0, ..., e_n$ are aggregate elements. Symbol $\#aggr$ is either $\#count$, $\#sum$, $\#max$, or $\#min$. Symbol $\prec$ 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 only see those literals outside of aggregate atoms.

Example 4.17 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.

4.2.3 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 4.18 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 that $r \geq l$, and $c_1; ...; c_n$ is a semi-colon 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 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 4.19 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.

4.2.4 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 the compatibility of this feature was not a focus of the system. When a function term is encountered by PREDICTOR, it simply sees the function term as an object constant.

4.2.5 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).

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

Example 4.20 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.

4.3 Language, Libraries, and Usage

We now specify the library and language details of PREDICTOR. System PREDICTOR is developed using the Python 3 programming language[3]. System CLINGO is an answer set programming toolkit that integrates grounder GRINGO (Gebser et al. 2007, 2010) and solver CLASP (Gebser et al. 2012) as a single system. PREDICTOR utilizes PYCLINGO[4] version 5, a Python API sub-system of CLINGO (Gebser et al. 2015). The PYCLINGO API enables users to easily access and enhance standard ASP features 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 is able to process the program step-by-step according to the procedures described above. System PREDICTOR also uses NETWORKX[5] version 2, a Python library that provides numerous operations for creating, modifying, and utilizing graphs. NETWORKX is utilized by PREDICTOR to perform all necessary graph operations, such as representing the dependency graph and simple component graph as well as computing strongly connected components and performing topological sorts.

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

[3]https://www.python.org/downloads/
[4]https://github.com/potassco/clingo
[5]https://networkx.github.io/

in Python 3, or it can be accessed through a command line interface. Figure 8 shows a simple example of how the package can be used within a Python program to generate a prediction of both the entire program as well as individual rules. Note that individual rules can also be constructed using Clingo's abstract syntax tree API [6]. Figure 9 shows a simple example of how PREDICTOR can be used through the command line interface. Here, the value to the --$predict\_rules$ flag, $file/with/rules.lp$, is a logic program containing rules whose grounding size is estimated under the assumption that these rules are appended towards the program $path/to/asp/program.lp$. The flag --key $q/2[0]$ specifies that $q/2[0]$ is a key predicate argument (note here we use 0-based indexing). The output is the sum of estimates for all rules in the file. Source code, system documentation, installation, and usage instructions for PREDICTOR are available here[7].

[6] http://potassco.sourceforge.net/doc/pyclingo/clingo.ast.html

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

```python
from predictor import Predictor
from pd_utils import Key, KeyList
from clingo_ast_util import PredicateSymbol

keys = KeyList([Key(PredicateSymbol('q', 2), [0])])
predictor = Predictor(keys)
predictor.load_program('path/to/asp/program.lp')
predictor.load_program('another/path/to/asp/program.lp')
# Both program files above will be used when generating predictions

# Predict the size of the entire loaded program
prog_size = predictor.predict_loaded_program()
print('The size of the loaded program is: %s' % prog_size)

rule = 'p(X) :- q(X,Y), r(X).'
# Predict the size of an individual rule
rule_size = predictor.predict_all(rule)
print('The size of the rule is: %s' % rule_size)

# Multiple rules may be used as well
multi_rules = '''
:- r(X), q(X).
:- r(X), t(X).
'''
multi_size = predictor.predict_all(multi_rules)
print('The size of all rules is: %s' % multi_size)
```

Figure 8: Example of using PREDICTOR as an imported library

```
predictor path/to/asp/program.lp --predict_rules file/with/rules.lp
    --key q/2[0]
```

Figure 9: Example of using PREDICTOR through the command line interface

5 System PROJECTOR Integration

In addition to developing PREDICTOR for this thesis, system PROJECTOR has been updated to interact with PREDICTOR. Figure 10 demonstrates how PREDICTOR is integrated with system PROJECTOR. We refer to the version of PROJECTOR integrated with PREDICTOR as PRD-PROJECTOR. Note how PREDICTOR runs entirely independent of and prior to the grounding step.

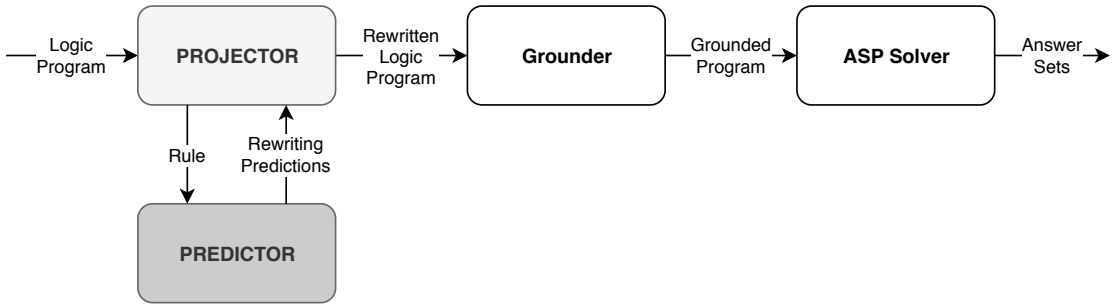

Figure 10: Typical ASP system architecture extended with PROJECTOR using PREDICTOR

Predicting the size of a projection is straightforward using PREDICTOR. We compute the predicted grounding size of a projection by taking the sum of predictions for the replacement rule and the projection rule.

System PRD-PROJECTOR uses PREDICTOR to make decisions on which projections, if any, to perform. In particular, it is used in two ways:

1. When PROJECTOR encounters a tie through its default heuristics for selecting variables to project, 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.

It is important to 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.

6 Experimental Analysis

To effectively 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. The goal of this evaluation is to determine how accurately PREDICTOR estimates the grounding size of rules. Second, an extrinsic evaluation of PRD-PROJECTOR is conducted. This evaluation examines the *relative* accuracy of system PREDICTOR, especially alongside PROJECTOR. In other words, it measures the quality of PREDICTOR by analyzing the impact it has on PROJECTOR rewritings. All benchmarks are gathered 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, NETWORKX version 2.3, and PYCLINGO version 5.4.0 are used to run PREDICTOR. Grounding and solving are also done by CLINGO version 5.4.0. For all benchmarks, execution was limited to 5 minutes.

Program Information Benchmarks were gathered on multiple programs from two different sources, and the same programs are used for both the intrinsic and extrinsic evaluation. 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 for benchmarking. The selection criteria was simple: if system PROJECTOR performed any projections on the program, it was selected.

For each program, 20 instances were evaluated. These instances, which are the same as the instances originally selected for the competition[8], were randomly selected for each program from a larger set of instances and feature varying levels of difficulty. 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 for an application called ASPCCG, an ASP-based natural language parser (Lierler & Schueller 2011). This program has been extensively studied by Buddenhagen and Lierler in (Buddenhagen & Lierler 2015) to evaluate the

[8]See https://www.mat.unical.it/aspcomp2014#Instance_Selection

impact of various rewritings on performance, of which one type of rewriting was projections. Program ASPCCG version 0.1 (ASPCCG-0.1) and ASPCCG version 0.2 (ASPCCG-0.2) encode the same problem, yet vary on how the ASP rules of the problem are specified. Even so, the difference in performance that accumulated from many rewritings in ASPCCG-0.2 is substantial over ASPCCG-0.1. Program ASPCCG-0.2 was derived from ASPCCG-0.1 by manually performing numerous rewritings and evaluating their effects, with grounding size and solving time being the driving measures behind the selected rewritings. The path from ASPCCG-0.1 to ASPCCG-0.2 consisted of 20 encodings.

This domain was used to evaulate system PROJECTOR in (Hippen & Lierler 2019). In their evaluation they considered 3 of the 20 encodings from ASPCCG. In order to continue these efforts, these same 3 encodings are considered in our evaluation, which we reiterate here:

- the ENC1 encoding that constitutes ASPCCG-0.1,
- the ENC7 encoding that constitutes one of the improved encodings on the path from ASPCCG-0.1 to ASPCCG-0.2, and
- the ENC19 encoding that constitutes ASPCCG-0.2.

Encoding ENC1 is demonstrative of a program without any effort put into optimization. Encoding ENC7 was selected as an encoding that represents a moderately optimized program. Encoding ENC19 is demonstrative of a program with significant professional optimizations performed. It is important to note that a significant amount of manual time and effort went into producing ENC7 and ENC19.

The instances for ASPCCG were gathered using sentences from CCGbank[9], a corpus of real-world sentences annotated with the combinatory categorical grammar (CCG) formalism. These sentences were separated by word count into five groups restricting sentences to those containing between 6 and 25 words. An equal number of sentences from each group were randomly selected to create the set of instances. The experiments for PROJECTOR utilized

[9] http://groups.inf.ed.ac.uk/ccg/ccgbank.html

the set of 60 instances that Buddenhagen and Lierler called the held-out set. These same 60 instances have been used in our evaluation.

Table 1 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).

Table 1: 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	c,b	**Yes**
ASPCCG ENC1	n,a,b,c,f	N/A
ASPCCG ENC7	n,a,b,c,f	N/A
ASPCCG ENC19	n,a,b,c,f	N/A

6.1 System PREDICTOR Accuracy

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. The *error factor* of a program instance can be computed by S'/S. The *average error factor* of a program is the average of all error factors across the instances of a program.

Table 2 shows the average error factor for all programs. The second column displays the average error factor using manually specified keys. The specific keys used are provided in Appendix A. These keys were identified only for root predicate arguments. The third column displays the average error factor without specifying keys. The average error factor shown was rounded to make comparisons easier. An asterisk ($*$) next to a program name indicates that not all 20 instances were grounded. Specifically, the *Incremental Scheduling* program was only able to ground 19 instances, while the *Permutation Pattern Matching* program could only ground 17 instances before timing out. In both cases we only report the numbers for instances with real grounding data.

Table 2: Average error factor for benchmark programs, with and without keys

Program	**Average Error Factor**	**Average Error Factor (Keyless)**
Hanoi Tower	1.5	1.5
Nomystery	1.5	1.5
Permutation Pattern Matching$*$	$\mathbf{3.8}$	5.0
Solitaire	4.3	4.3
Stable Marriage	$\mathbf{3.7}$	$7.5 * 10^5$
Bottle Filling	$4.9 * 10^9$	$4.9 * 10^9$
Incremental Scheduling$*$	$1.1 * 10^5$	$1.1 * 10^5$
Labyrinth$*$	$1.3 * 10^1$	$1.3 * 10^1$
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	$1.9 * 10^{-4}$	$1.9 * 10^{-4}$
Ricochet Robots	$\mathbf{2.0 * 10^{-1}}$	$2.2 * 10^{-1}$
Weighted Sequence	$\mathbf{6.0 * 10^{-3}}$	$1.1 * 10^{-2}$

We partition the results into 3 sets using the average error factor with keys. First, there are 5 programs where the estimates computed by PREDICTOR are, on average, less than 1 order of magnitude off. These programs are: *Hanoi Tower*, *Nomystery*, *Permutation Pattern Matching*, *Solitaire*, and *Stable Marriage*. Second, there are 8 programs that are, on average, over 1 order of magnitude off for predictions. These programs are: *Bottle Filling*, *Incremental Scheduling*, *Labyrinth*, *Minimal Diagnosis*, *Valves Location*, and all 3

encodings from ASPCCG. Finally, 3 programs are predicted to have lower grounding sizes than reality. These programs are: *Knight Tour with Holes*, *Ricochet Robots*, and *Weighted Sequence*.

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.

Overall, 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 six main reasons for the poor accuracy of 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. Recall Example 4.11. We reiterate our final statements of the example here. While we were able to estimate that the argument size of $q[1]$ is 2, this is entirely due to our data distribution handling for numeric data. Had we considered the object constants in the program as non-numeric, we would have estimated that the argument size of $q[1]$ is instead 4. Unfortunately, most programs have object constants that are non-numeric. Even if there are numeric constants, they may not be uniformly distributed between minimum and maximum values. PREDICTOR is not at this point able to model this data properly.

2. Since we only identified keys for root predicate arguments, many keys were likely missed. It is reasonable to believe that some benchmarks could see great benefits from better key selection.

3. System PREDICTOR has limited support for certain language extensions, which we discussed in Section 4.2. Programs containing these features may be prone to more faulty results. Aggregates in particular are a common feature that can noticeably impact grounding size.

4. Non-tight programs are not modeled well at this point in PREDICTOR. 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. Consider the following rule from *Knight Tour with Holes*:

$$number(X-1) \leftarrow number(X), 1 < X.$$

Rules of this pattern, where a value is recursively updated using an incremental or decremental binary operation term, are not modeled well with PREDICTOR. *Weighted Sequence* and *Ricochet Robots* features very similar rules to the one above.

5. System PREDICTOR is vulnerable to what is known as *error propagation*. If our estimates were computed only from known statistics (i.e. root predicate arguments), estimates would often not be as erroneous. Unfortunately, almost all programs will require generating estimates from other estimates. Erroneous argument size estimates of predicates will make those predicates that depend on them be erroneous, typically more so, too. This results in exponentially worse argument size estimates from predicates arguments further from root predicate arguments in the argument dependency graph. This phenomenon, known as error propagation, has also been observed with join size estimations in relational database systems when many joins are used in a query (Ioannidis & Christodoulakis 1991).

6. Finally, we do not account for any built-in rewriting performed by GRINGO when computing error factor. Most grounders, in addition to performing the necessary steps for intelligently grounding the program, will internally perform some rewritings to improve the program. This makes it possible that the grounding size reported by GRINGO does not necessarily reflect the exact program we used for predictions. This issue is largely unavoidable, as rewritings performed will vary depending on the

grounder chosen (e.g. optimized grounding processes in grounder IDLV (Calimeri et al. 2017)).

6.2 Evaluation of PRD-PROJECTOR

Despite the accuracy of PREDICTOR often being rather poor, demonstrated by the average error factor on a set of benchmarks, these predictions may still be able to help determine whether or not a rewriting is worth performing. This can occur because predictions may still properly determine which of two sets of rules produce a smaller grounding.

Let S be the grounding size of an instance in a program computed by GRINGO. Let S' be the grounding size of the same instance in a modified version of the program computed by GRINGO. 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 can be computed by S'/S. As such, a grounding size factor greater than 1 indicates that the modification increased the grounding size, whereas if it is less than 1 it indicates that the modification improved/decreased the grounding size. Naturally, the *average grounding size factor* of a program is the average of all grounding size factors across the instances of a program.

Table 3 displays the average grounding size factor for PROJECTOR (column 2) and PRD-PROJECTOR (column 3) on all benchmark programs. Like in Table 2, an asterisk ($*$) following a program name indicates that not all 20 instances were grounded. For the *Incremental Scheduling* program, only 19 instances were grounded. The same instances could not be grounded with the rewritten programs. The *Permutation Pattern Matching* program was only able to ground 17 instances. It is interesting to note that both the program rewritten with PROJECTOR and the program rewritten with PRD-PROJECTOR grounded all 20 instances. 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

very slight improvement for PRD-PROJECTOR, however this information was lost for the precision shown.

Table 3: Average grounding size factor of PROJECTOR and PRD-PROJECTOR

Program	PROJECTOR	PRD-PROJECTOR
Hanoi Tower	1.41	1.00
Incremental Scheduling∗	1.14	1.09
Minimal Diagnosis	1.06	1.00
Solitaire	1.41	1.00
Stable Marriage	0.13	0.11
ASPCCG ENC1	0.63	0.52
ASPCCG ENC7	1.40	1.24
ASPCCG ENC19	1.58	0.97
Bottle Filling	1.36	1.36
Labyrinth∗	1.11	1.11
Permutation Pattern Matching∗ †	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

We partition the results into three sets. First, there are 8 programs in which PRD-PROJECTOR reduces the grounding size noticeably when compared to PROJECTOR. These programs are: *Hanoi Tower*, *Incremental Scheduling*, *Minimal Diagnosis*, *Solitaire*, *Stable Marriage*, and all 3 encodings of ASPCCG. Next, there are 5 programs in which PRD-PROJECTOR does not impact the grounding size noticeably when compared to PROJECTOR (although for three of the programs there are still very slight improvements). These programs are: *Bottle Filling*, *Labyrinth*, *Permutation Pattern Matching*, *Valves Location*, and *Weighted Sequence*. Finally, there are 3 programs in which PRD-PROJECTOR increased the grounding size noticeably when compared to PROJECTOR. These programs are: *Knight Tour with Holes*, *Nomystery*, and *Ricochet Robots*.

There are a couple of interesting takeaways from these results. First, they suggest that PREDICTOR often does well at comparing the grounding sizes of rules and determining which rules will produce smaller groundings, regardless of the accuracy of predictions for

the entire program. Even though the results are mostly positive, they do demonstrate that it is possible that PREDICTOR can mislead a rewriting system, producing grounding sizes that are larger than they would have without it.

The *execution time* of a program is the amount of time it takes to ground and solve a program. While execution time was not a focus of this thesis, it is still ultimately what is important to programmers. We present benchmarks related to execution time in Table 5 of Appendix B and summarize the results here. Of the 8 programs in which PRD-PROJECTOR reduces the grounding size noticeably when compared to PROJECTOR, only *Incremental Scheduling* has a higher execution time compared to PROJECTOR. The 3 programs whose grounding size slightly improved for PRD-PROJECTOR (indicated by a dagger symbol (†) on Table 3) also saw an improvement or no perceivable change to their execution time compared to PROJECTOR. Naturally, programs *Bottle Filling* and *Labyrinth*, whose rewritten program is the same between PROJECTOR and PRD-PROJECTOR, did not have a change in execution time. Finally, of the three programs whose grounding size increased noticeably when compared to PROJECTOR, only *Nomystery* has a lower execution time compared to PROJECTOR.

Results of ASPCCG We now take a closer look at the benchmarks for ASPCCG as a continuation to the efforts by Hippen and Lierler in (2019). There, they found that when PROJECTOR was ran on ENC1, it consistently reduced the grounding size. However, when PROJECTOR was ran on ENC7, it slightly increased the grounding size for all but two instances. When PROJECTOR was ran on ENC19, it increased the grounding size noticeably across all instances. These results are consistent with our findings in Table 3.

For a more detailed view of the grounding size factor at the instance level, we refer the read to Figures 11, 12, and 13 in Appendix C. These graphs display grounding size factors for each instance of ENC1, ENC7, and ENC19 respectively. Across all three encodings, there is only a single instance in ENC19 where PRD-PROJECTOR produces a larger grounding size than PROJECTOR, and in that case the grounding size was only slightly larger. We also

see that while PRD-PROJECTOR matches the same narrative as PROJECTOR for ENC1 and ENC7, PRD-PROJECTOR actually lowers the grounding size or only slightly increases the grounding size in all but 4 instances for ENC19. Finding optimal rewritings in ENC19 is especially notable since this encoding is already very well optimized.

7 Conclusions and Future Work

While many automated rewriting systems for non-ground ASP logic programs exist, not all of these systems provide a guarantee that their rewritings will produce a better program. One thing that many of these systems share, however, is that they typically attempt to reduce the grounding size of a program. Indeed, grounding size often correlates with solving time (Gebser et al. 2011). Even so, grounding a program is an expensive task by itself, making it non-viable to use the real grounding size as a guiding metric for solving time in automated rewriting systems.

In this thesis we explore a solution to this issue. We introduce a new system, called PREDICTOR, meant to estimate grounding sizes for ASP logic programs. To achieve this we utilized methods from join-order size estimations in relational databases. System PREDICTOR can run independent of the chosen grounding and solving system. Furthermore, PREDICTOR is capable of running on many real programs following ASP-Core-2 standards, including both tight and non-tight programs.

We also extend the automated ASP rewriting system called PROJECTOR to utilize the estimations generated by PREDICTOR as a guide for rewriting decisions. Our evaluation consists of two methodologies. First, we conduct an intrinsic assessment of PREDICTOR by measuring the accuracy of our estimations compared to the real grounding size produced by ASP grounder GRINGO. Here we find that PREDICTOR estimations are often very inaccurate, although it is capable of producing reasonable estimations for certain programs. We then perform an extrinsic evaluation where we use the extended PROJECTOR system to measure the grounding sizes of programs compared to original PROJECTOR. The results from this

evaluation suggest that PREDICTOR is useful as a guide for automated rewriting systems.

The results of this thesis open up several areas of direction for future work:

- *Improve data modeling* The data modeling in PREDICTOR is only useful for numeric object constants that are distributed mostly incrementally between the minimum and maximum values. Improving support for non-numeric data may lead to more accurate estimations.
- *Automatically identify keys* Currently, the user must manually specify keys for PREDICTOR. This is somewhat contradictory to the goal of many automated rewriting systems, where one of the goals may be to ensure the user does not need to perform any extra work. Furthermore, manual identification of keys are prone to human error. Automatically determining which indices are a primary key of the predicate will fix this issue.
- *Improve language support* While PREDICTOR is capable of running on many ASP-Core-2 compliant programs, there are still programs that PREDICTOR cannot run well on, detailed in Section 4.2. For example, function terms are seen as object constants which can lead to missing variables in the rule.
- *Improve non-tight program support* Currently the non-tight program argument size estimations in PREDICTOR can lead to drastic underestimations. Improving the estimations so that it accounts for the potential number of cycles that could occur before a fixed point in the grounded program is reached during normal grounding processing will help eliminate this issue.
- *Expand grounding size meaning beyond the number of rules* As explained in Section 4.2, aggregate and choice rules can lead to the length of a rule expanding during grounding. It may be useful to measure this expansion as well.

Overall, we believe that this work sets a strong foundation for developing future methods in estimating grounding sizes prior to any grounding processes.

A Key Information

Table 4 shows the keys used for each benchmark program. The key naming is formatted as $predicateName/arity[indices]$ where $predicateName$ is the name of the predicate on which we are creating a key, $arity$ is the arity of the predicate, and $indices$ is a comma separated list of indices to include as part of the key.

Table 4: Key information for benchmark programs

Program	**Keys**
Bottle Filling	-
Hanoi Tower	-
Incremental Scheduling	$precedes/2[0]$, $importance/2[0]$, $job\_device/2[0]$, $job\_len/2[0]$, $deadline/2[0]$, $curr\_job\_start/2[0]$, $curr\_on\_instance/2[0]$, $instances/2[0]$
Knight Tour with Holes	-
Labyrinth	-
Minimal Diagnosis	$obs\_elabel/3[0,1]$
Nomystery	$at/2[0]$, $fuel/2[0]$, $goal/2[0]$
Permutation Pattern Matching	$t/2[0]$, $p/2[0]$
Ricochet Robots	$amo/2[0]$, $d1/2[0]$, $dir/2[0]$
Solitaire	-
Stable Marriage	$manAssignsScore/3[0,1]$, $womanAssignsScore/3[0,1]$
Valves Location	$dem/3[0,1]$
Weighted-Sequence	$leafWeightCardinality/3[0]$
ASPCCG ENC1	$word\_at/2[1]$, $category\_tag\_nofeatures/3[0]$, $category\_tag/3[0]$, $adjacent/2[0]$
ASPCCG ENC7	$word\_at/2[1]$, $category\_tag\_nofeatures/3[0]$, $category\_tag/3[0]$, $adjacent/2[0]$
ASPCCG ENC19	$word\_at/2[1]$, $category\_tag\_nofeatures/3[0]$, $category\_tag/3[0]$, $adjacent/2[0]$

B Execution Times

Let S be the execution time of an instance in a program computed by CLINGO. Let S' be the execution time of the same instance in a modified version of the program computed by CLINGO. 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

execution time factor of a program's instance can be computed by S'/S. The *average grounding size factor* of a program is the average of all grounding size factors across the instances of a program. Table 5 shows the *average execution time factor* of programs rewritten with PROJECTOR and PRD-PROJECTOR. Unlike Table 3, an additional column is present that indicates the minimum number of instances solved for all 3 versions of the program (original, PROJECTOR, PRD-PROJECTOR). The average execution time is only computed from instances with solve times for all 3 versions of the program.

Table 5: Average execution time factor of PROJECTOR and PRD-PROJECTOR

Program	**Solved**	PROJECTOR	PRD-PROJECTOR
Hanoi Tower	20	1.67	1.00
Incremental 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
Permutation Pattern Matching	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

We direct the reader to Table 3 for benchmarks on the grounding size factor of PROJECTOR and PRD-PROJECTOR. Of the 8 programs in which PRD-PROJECTOR reduces the grounding size noticeably when compared to PROJECTOR, only *Incremental Scheduling* has a higher execution time compared to PROJECTOR. The 3 programs whose grounding size slightly improved for PRD-PROJECTOR (indicated by a dagger symbol (†) on Table 3) also saw an improvement or no perceivable change to their execution time compared to PROJECTOR. Naturally, programs *Bottle Filling* and *Labyrinth*, whose rewritten program is the same between PROJECTOR and PRD-PROJECTOR, did not have a change in execution time.

Finally, of the three programs whose grounding size increased noticeably when compared to PROJECTOR, only *Nomystery* has a lower execution time compared to PROJECTOR.

C Full ASPCCG Benchmarks

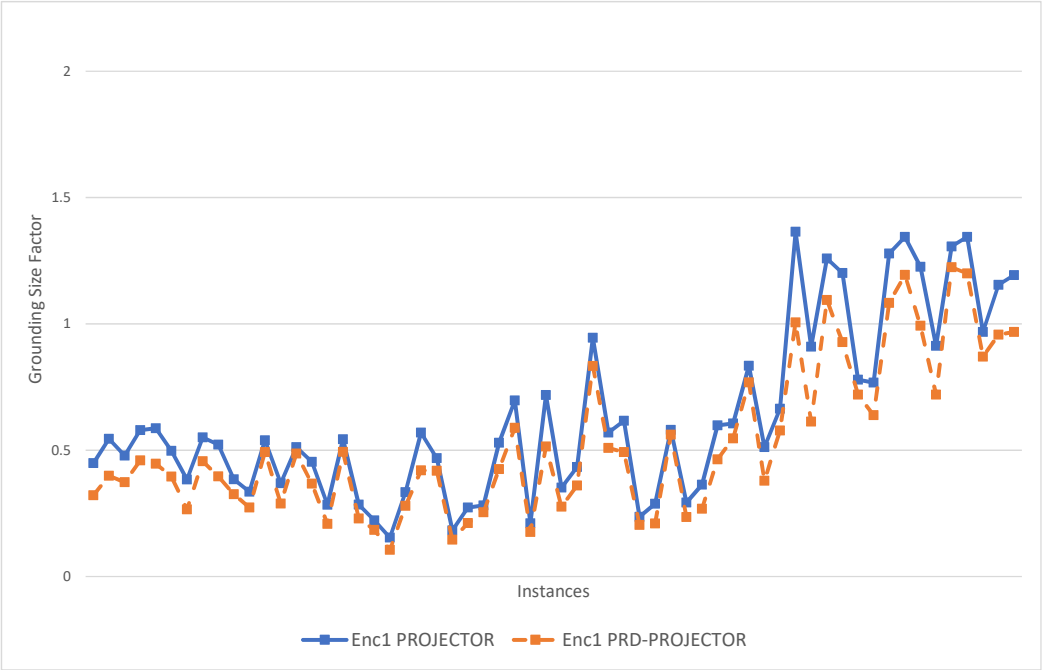

Figure 11: Grounding size factor for instances of ENC1

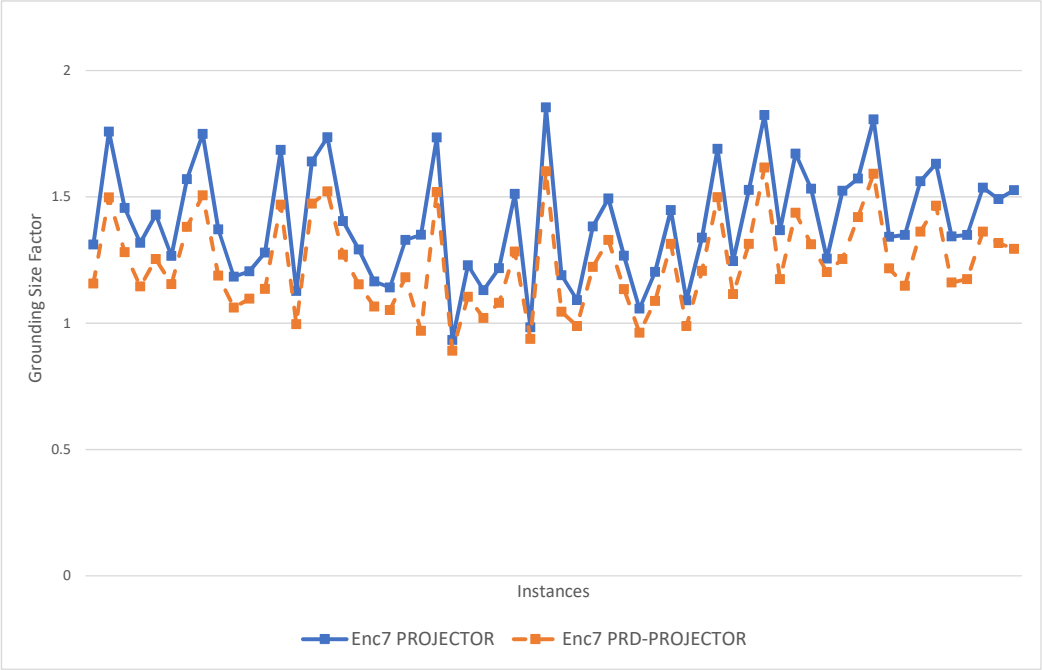

Figure 12: Grounding size factor for instances of ENC7

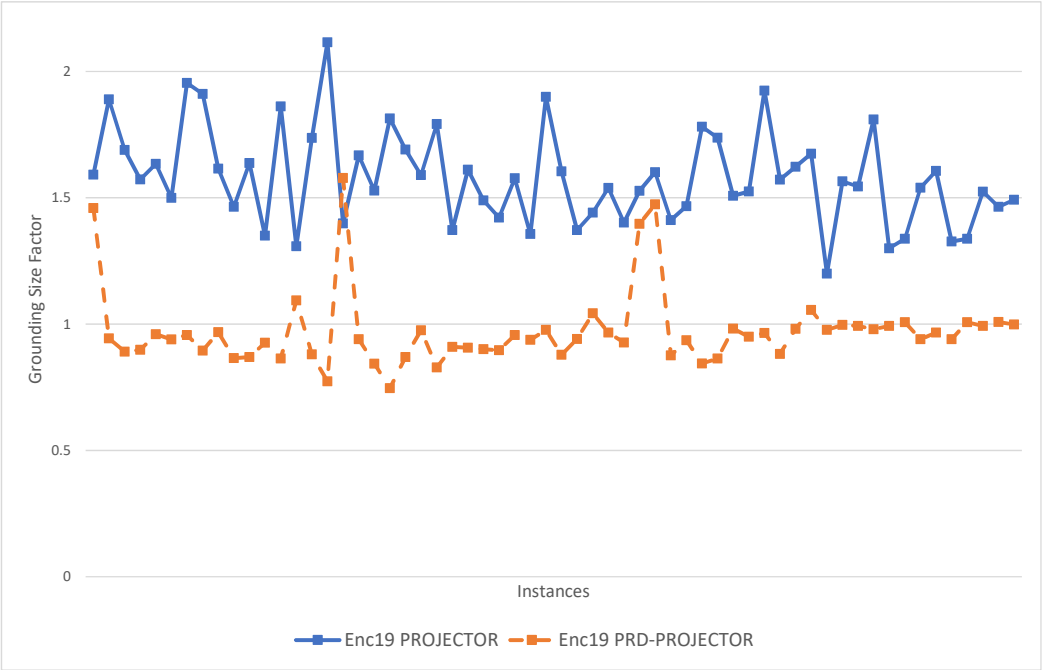

Figure 13: Grounding size factor for instances of ENC19

References

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

Bichler, M., Morak, M. & Woltran, S. (2016), lpopt: A rule optimization tool for answer set programming, *in* 'Proceedings of International Symposium on Logic-Based Program Synthesis and Transformation'.

Brewka, G., Eiter, T. & Truszczynski, M. (2011), 'Answer set programming at a glance', *Commun. ACM* **54**(12), 92–103.

Buddenhagen, M. & Lierler, Y. (2015), Performance tuning in answer set programming, *in* 'Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)'.

Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Ricca, F. & Schaub, T. (2012), 'Asp-core-2: Input language format', *ASP Standardization Working Group* .

Calimeri, F., Fusca, D., Perri, S. & Zangari, J. (2017), 'I-dlv: The new intelligent grounder of dlv', *Intelligenza Artificiale* **11**(1), 5–20.

Calimeri, F., Fusca, D., Perri, S. & Zangari, J. (2018), Optimizing answer set computation via heuristic-based decomposition, *in* 'International Symposium on Practical Aspects of Declarative Languages', Springer, pp. 135–151.

Calimeri, F., Gebser, M., Maratea, M. & Ricca, F. (2016), 'Design and results of the fifth answer set programming competition', *Artificial Intelligence* **231**, 151 – 181.
URL: *http://www.sciencedirect.com/science/article/pii/S0004370215001447*

Eiter, T., Fink, M., Tompits, H., Traxler, P. & Woltran, S. (2006), Replacements in non-ground answer-set programming, *in* 'Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)'.

Eiter, T., Traxler, P. & Woltran, S. (2006), An implementation for recognizing rule replacements in non-ground answer-set programs, *in* 'Proceedings of European Conference On Logics In Artificial Intelligence (JELIA)'.

Faber, W., Leone, N., Mateis, C. & Pfeifer, G. (1999), Using database optimization techniques for nonmonotonic reasoning, pp. 135–139.

Faber, W., Leone, N. & Perri, S. (2012), The intelligent grounder of dlv, *in* 'Correct Reasoning', Springer, pp. 247–264.

Gebser, M., Kaminski, R., Kaufmann, B., Lindauer, M., Ostrowski, M., Romero, J., Schaub, T. & Thiele, S. (2015), 'Potassco user guide', *Institute for Informatics, University of Potsdam, second edition edition* .

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T. & Thiele, S. (2010), A user's guide to gringo, clasp, clingo, and iclingo. Available at `http://potassco.sourceforge.net`.

Gebser, M., Kaminski, R., Kaufmann, B. & Schaub, T. (2011), Challenges in answer set solving, *in* M. Balduccini & T. Son, eds, 'Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays in Honor of Michael Gelfond', Vol. 6565, Springer, pp. 74–90.

Gebser, M., Kaufmann, B. & Schaub, T. (2012), 'Conflict-driven answer set solving: From theory to practice', *Artificial Intelligence* **187**, 52–89.

Gebser, M., Schaub, T. & Thiele, S. (2007), Gringo: A new grounder for answer set programming, *in* 'Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning', pp. 266–271.

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

Ioannidis, Y. E. & Christodoulakis, S. (1991), On the propagation of errors in the size of join results, Technical report, University of Wisconsin-Madison Department of Computer Sciences.

Lierler, Y. (2017), 'Algorithms in backtracking search behind sat and asp', https://works.bepress.com/yuliya_lierler/73/.

Lierler, Y., Maratea, M. & Ricca, F. (2016), 'Systems, engineering environments, and competitions', *AI Magazine* **37**(3).

Lierler, Y. & Schueller, P. (2011), 'Parsing combinatory categorial grammar with answer set programming: Preliminary report'.
URL: *http://www.cs.utexas.edu/users/ai-lab/pub-view.php?PubID=127116*

Rossi, F., van Beek, P. & Walsh, T. (2008), Constraint programming, *in* F. van Harmelen, V. Lifschitz & B. Porter, eds, 'Handbook of Knowledge Representation', Elsevier, pp. 89–134.

Silberschatz, A., Korth, H. F., Sudarshan, S. et al. (1997), *Database system concepts*, Vol. 4, McGraw-Hill New York.

Syrjänen, T. (2000), 'Lparse 1.0 user's manual'.