Automatic Program Rewriting in Non-Ground Answer Set Programs *

Nicholas Hippen and Yuliya Lierler

University of Nebraska Omaha
{nhippen,ylierler}@unomaha.edu

Abstract. Answer set programming is a popular constraint programming paradigm that has seen wide use across various industry applications. However, logic programs under answer set semantics often require careful design and nontrivial expertise from a programmer to obtain satisfactory solving times. In order to reduce this burden on a software engineer we propose an automated rewriting technique for non-ground logic programs that we implement in a system PROJECTOR. We conduct rigorous experimental analysis, which shows that applying system PROJECTOR to a logic program can improve its performance, even after significant human-performed optimizations.

1 Introduction

Answer set programming (ASP) [4] is a leading knowledge representation/declarative programming paradigm. ASP seeks to provide techniques and tools to quickly and reliably design robust software solutions for complex knowledge-intensive applications. It reduces the programming task to modeling an application domain as a set of logic rules, and leaves all computational concerns to automated reasoning. Many efficient implementations of automated reasoning tools for ASP that include grounders and solvers are available. See [19] for a brief survey of grounders – e.g., LPARSE, GRINGO, IDLV– and solvers – e.g., SMODELS, DLV, CLASP. Thanks to these implementations, ASP has been successfully used in scientific and industrial applications. Examples include decision support systems for space shuttle flight controllers [1] and team building and scheduling [21].

These successful applications notwithstanding, ASP faces challenges. Practice shows that to achieve a required level of performance it is *crucial* to select the *right combination* of a representation and a processing tool. Unfortunately, at present this fundamental task is still not well understood, and it typically requires substantial expertise and effort, and there is still no guarantee of success. Gebser et al. in [14] presented a set of "rules-of-thumb" used by their expert team in tuning ASP solutions. These rules include suggestions on program rewritings that often result in substantial performance gains. Buddenhagen and Lierler in [5] studied the impact of the rewritings on an ASP-based natural language parser called ASPCCG [20]. They reported orders of magnitude

* We are grateful to Michael Dingess, Brian Hodges, Daniel Houston, Roland Kaminski, Liu Liu, Miroslaw Truszczynski, Stefan Woltran for the fruitful discussions.

in gains in memory and time consumption as a result of some program transformations they executed manually. One of the rewriting techniques used in that application a number of times was so called "projection". In this paper, we present a system that performs various forms of projection automatically. We then extensively study the effects of automatic projection. In particular, we revisit several versions of an ASP-based natural language parser ASPCCG and evaluate the effects of different variants of projection on their performance. We also consider several benchmarks from the Fifth Answer Set Programming Competition.

Related Work The possible impact of program rewritings on its performance is well understood. Many answer set solvers start their computation by performing propositional program simplifications based on answer set preserving rewritings, see, for instance, [18, Section 6.1] and [15]. It is important to note that solvers will perform their preprocessing on ground programs. Tool SIMPLIFY[1] [10, 11] implements two program simplification techniques for non-ground disjunctive programs, namely, rule subsumption (dropping a rule in presence of another "subsuming" rule) and shifting (replacing a rule with disjunction in its head by rules without disjunction, when possible). System LPOPT[2] [2, 3] decomposes rules of an ASP program, in the following way. Given a rule to rewrite, LPOPT may replace it with several new ones with a guarantee that the number of distinct variables occurring in each of these rules is less than that of the original. This is done by constructing a graph problem based on a given rule. A general-purpose library called *htd*[3] is used to find a solution to such graph problem. A resulting solution is then used by LPOPT to compose logic rules to replace the given one. In this work we continue the efforts undertaken by LPOPT, and propose a system called PROJECTOR. Unlike LPOPT, PROJECTOR develops its own rewriting strategies rooted in ideas underlying projection technique – a database optimization technique – commonly used by ASP practitioners in optimizing their encodings as well as ASP grounders [12, 7].

Paper Outline We start by presenting the notions of α and β-projection in Section 2. We continue into describing an algorithm called `Projection` for performing the rewritings of logic programs based on the ideas of α and β-projection. We then present the details behind system PROJECTOR that implements the `Projection` procedure. We conclude with the section on experimental analysis.

2 Projections In Theory

In this section we present two methods for program rewritings, namely α and β-projecting. We start by presenting some preliminary terminology and notation. We then proceed towards defining α and β-projecting for arbitrary logic rules.

Preliminaries We consider a vocabulary of function and predicate symbols associated with an arity (nonnegative integer). A function symbol of arity 0 is called a constant. A *term* is either a constant, a variable, or an expression of the form $f(t_1, \ldots, t_k)$

[1] `http://www.kr.tuwien.ac.at/research/systems/eq/simpl/index.html`

[2] `http://dbai.tuwien.ac.at/research/project/lpopt/`

[3] `https://github.com/mabseher/htd`

where f is a function symbol of arity $k > 0$ and t_i is a term. An *atom* has the form $p(t_1, \ldots, t_k)$ where p is a predicate symbol of arity k and t_i is a term. For instance, an atom $p(q(A), B, C, 1)$ is such that

- symbols $A,\ B,\ C$ are variables (we use the convention customary in logic programming and denote variables by identifiers starting with a capital letter),
- 1 is a constant (a function symbol of arity 0),
- q is a function symbol of arity 1, and
- p is a predicate symbol of arity 4.

A *rule* is an expression of the form

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

where $n \geq m \geq 0$, a_0 is either an atom or symbol $\bot$, and $a_1, \ldots, a_n$ are atoms. The atom a_0 is the *head* of the rule and $a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n$ is the *body*. At times we use letter $\mathbb{B}$ to denote a body of a rule. We call atoms and expressions of the form $not\ a$ (where a is an atom) *literals*. It is often convenient to identify the body of a rule with the set of the literals occurring in it. For example, we may identify the body of rule (1) with the set $\{a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n\}$. To literals $a_1, \ldots, a_m$ we refer as positive, whereas to literals $not\ a_{m+1}, \ldots, not\ a_n$ we refer as negative. We say that a *rule* is *positive* when its body consists only of positive literals. For example, the rule below is positive

$$p(A, D) \leftarrow q(A, B, C), r(A, D). \tag{2}$$

For a set L of literals, we say that a variable V is *unsafe* in L, if V does not occur in a positive literal in L. For instance, B is the only unsafe variable in $\{p(A),\ not\ q(A, B)\}$. We call a *rule safe* when no variable in this rule is an unsafe variable in its body. Rule (2) is safe. Rules $p(A).$ and $\bot \leftarrow p(A),\ not\ q(A, B).$ exemplify unsafe rules. In answer set programming, rules are required to be safe by the grounders [6].

α-projecting for positive rules Given a literal l by $vars[l]$ we denote the set of variables occurring in l. For example, $vars[p(f(A), B, C, 1)] = \{A, B, C\}$. Also, for a set L of literals $vars[L]$ denotes the set of all variables occurring in the elements of L. For instance, $vars[\{p(f(A), B, C, 1), r(A, D)\}] = \{A, B, C, D\}$.

For a rule ρ and a set V of variables, by $\alpha(\rho, V)$ we denote the set of all literals in the body of ρ such that they contain *some* variable in V. Let ρ_1 be a rule (2). Then,

$$\alpha(\rho_1, \{B\}) = \{q(A, B, C)\}$$
$$\alpha(\rho_1, \{B, C\}) = \{q(A, B, C)\}$$

For a literal l and a set V of variables, we say that l is V-free when no variable in V occurs in l. Symbol $\bot$ is V-free for any set V.

For a set V of variables and a positive rule ρ of the form $a \leftarrow \mathbb{B}$ where a is V-free, the process of *α-projecting V* out of this rule will result in replacing it by two rules:

1. a rule
$$q(\mathbf{t}) \leftarrow \alpha(\rho, V).$$
so that
   - q is a fresh predicate symbol with respect to original program, and

- $\mathbf{t}$ is composed of the variables that occur in $\alpha(\rho, V)$, but not in V (in other words, $\mathbf{t} = vars[\alpha(\rho, V)] \setminus V$; here we abuse the notation and associate a set of elements with a tuple. Let us assume a lexicographical order as a default order of elements in a constructed tuple — we will use this convention in the remainder of the paper);

2. a rule

$$a \leftarrow (\mathbb{B} \setminus \alpha(\rho, V)) \cup \{q(\mathbf{t})\}.$$

For instance, the result of α-projecting variable B (here we identify a variable with a singleton set composed of it) from ρ_1 follows:

$$\begin{array}{l} q'(A, C) \leftarrow q(A, B, C). \\ p(A, D) \leftarrow q'(A, C), r(A, D). \end{array}$$

The result of α-projecting variables $\{B, C\}$ from ρ_1 follows:

$$\begin{array}{l} q'(A) \leftarrow q(A, B, C). \\ p(A, D) \leftarrow q'(A), r(A, D). \end{array} \tag{3}$$

An order of projecting We associate projections with a positive integer n – *an order* – where n is the cardinality of $\alpha(\rho, V)$. For instance, the result of projecting variable B or variables $\{B, C\}$ from ρ_1 are projections of order 1. In other words, these are projections that affect only variables that occur in a single literal in a body of a rule. Let ρ_2 be a rule

$$p(A, D, F) \leftarrow q(A, B, C), r(B, D), s(D, E), u(C), w(F). \tag{4}$$

Then, $\alpha(\rho_2, \{B\}) = \{q(A, B, C), r(B, D)\}$. The result of α-projecting variable B from ρ_2 follows. This is an example of projection of order 2.

$$qr'(A, C, D) \leftarrow q(A, B, C), r(B, D). \tag{5}$$

$$p(A, D, F) \leftarrow qr'(A, C, D), s(D, E), u(C), w(F). \tag{6}$$

On grounders GRINGO **and** IDLV In practice, grounders GRINGO and IDLV implement instances of projection of order 1. For instance, given a program with rule ρ_1, IDLV rewrites this rule and replaces it with rules listed in (3). System GRINGO is capable to do the same rewriting when anonymous variables are used. For example, if a rule ρ_1 is stated as

$$p(A, D) \leftarrow q(A, \_, \_), r(A, D).$$

then GRINGO will replace it with rules listed in (3), where B and C are substituted by anonymous variable symbol _.

α-projecting for arbitrary rules We now generalize α-projecting to arbitrary rules. The reader may observe that the definitions become more complex. The complexity is due to the necessity of producing safe rules as a result of projecting. Recall that rules are required to be safe by the grounders.

For a set L of literals, L^u denotes the set of all unsafe variables in L. For function $\alpha(\rho, V)$, we define function $V\!\uparrow$ as follows

$$V\!\uparrow\!0 = V$$

and for $i = 0, 1, 2, \ldots$

$$V{\uparrow}i+1 = V{\uparrow}i \cup \alpha(\rho, V{\uparrow}i)^u.$$

For instance, consider extending rule (2) as follows

$$p(A, D) \leftarrow q(A, B, C), r(A, D), t(E), not\ s(B, E).$$

By ρ_3 we denote this rule. Then,

$$\alpha(\rho_3, \{B\}) = \{q(A, B, C), not\ s(B, E)\}$$
$$\{q(A, B, C), not\ s(B, E)\}^u = \{E\}$$
$$\{B\}{\uparrow}0 = \{B\}$$
$$\{B\}{\uparrow}1 = \{B, E\}$$
$$\{B\}{\uparrow}\omega = \{B, E\}$$
$$\alpha(\rho_3, \{B\}{\uparrow}\omega) = \{q(A, B, C), not\ s(B, E), t(E)\}.$$

For a set V of variables and a rule ρ of the form $a \leftarrow \mathbb{B}$ where a is V-free, the process of α-*projecting* V out of this rule will result in replacing it by two rules:

1. a rule
$$q(\mathbf{t}) \leftarrow \alpha(\rho, V{\uparrow}\omega)$$
so that q is a fresh predicate symbol with respect to original program and tuple $\mathbf{t}$ is composed of the variables that occur in $\alpha(\rho, V{\uparrow}\omega)$, but not in V;
2. a rule
$$a \leftarrow (\mathbb{B} \setminus \alpha(\rho, V{\uparrow}\omega)) \cup \{q(\mathbf{t})\}. \tag{7}$$

It is easy to see that for a positive rule, $\alpha(\rho, V{\uparrow}\omega) = \alpha(\rho, V)$. Thus, the presented definition of α-projecting for an arbitrary rule is a generalization of this concept for positive rules. The result of α-projecting B from ρ_3 follows:

$$qst'(A, C, E) \leftarrow q(A, B, C), t(E), not\ s(B, E).$$
$$p(A, D) \leftarrow r(A, D), qst'(A, C, E).$$

β-projecting For a rule ρ and a set V of variables, by $\beta(\rho, V{\uparrow}\omega)$ we denote the set of all literals in the body of ρ such that *all* their variables are contained in $vars[\alpha(\rho, V{\uparrow}\omega)]$.

For example,

$$\beta(\rho_2, \{B\}{\uparrow}\omega) = \{q(A, B, C), r(B, D), u(C)\}$$
$$\beta(\rho_3, \{B\}{\uparrow}\omega) = \alpha(\rho_3, \{B\}{\uparrow}\omega)$$

It is easy to see that $\alpha(\rho, V{\uparrow}\omega) \subseteq \beta(\rho, V{\uparrow}\omega)$.

For a set V of variables and a rule ρ of the form $a \leftarrow \mathbb{B}$ where a is V-free, the process of β-*projecting* V out of this rule will result in replacing it by two rules:

1. a rule
$$q(\mathbf{t}) \leftarrow \beta(\rho, V{\uparrow}\omega)$$
so that q is a fresh predicate symbol with respect to original program and tuple $\mathbf{t}$ is composed of the variables that occur in $\beta(\rho, V{\uparrow}\omega)$, but not in V;
2. a rule (7).

For instance, the result of β-projecting variable B from rule ρ_2 consists of a rule

$$qr'(A, C, D) \leftarrow q(A, B, C), r(B, D), u(C). \tag{8}$$

and rule (6). Note that rules (5) and (8) differ only in one atom in the body, namely, $u(C)$. The result of β-projecting variable B from rule ρ_3 coincides with that of α-projecting B from ρ_3.

In the sequel, it is convenient for us to refer to the first rule produced in α and β-projecting as α and β-rules respectively. For instance, rule (5) is the α-rule of α-projecting variable B from ρ_2. The second rule produced in α and β-projecting we call a *replacement* rule.

3 Projections in Practice

In this section we describe an algorithm, which carries out program rewritings utilizing α and β-projection. We implement this algorithm in a system called PROJECTOR[4]. We conclude this section by providing technical details on the PROJECTOR implementation.

Algorithm 1 presents procedure `Projection`. As an input it takes a rule ρ, a projection type τ (that can take values α or β), and a positive integer n that specifies the order of projecting. Algorithm `Projection` performs projection iteratively. It picks a set of variables for projecting and computes the respective τ and replacement rules. It then attempts to repeat the same procedure on the replacement rule. Procedure `Projection` starts by defining V as the set of variables on which the τ-projecting of order n or less may be applied (line 2). Next, $\mathbb{R}$ is initialized as an empty set that will in the future hold rules added by projection. Lines 5-11 handle variable selection so that set W computed in these lines contains variables to project on. This selection process targets to perform projections of smaller orders first. Also, this process groups any variables that can be projected together without introducing new literals into the τ-rule. Then, the τ-rule is computed (line 12). If the set of variables occurring in the body of the computed τ-rule *is different* from the set of variables occurring in the body of ρ, we add the τ-rule to $\mathbb{R}$ (line 14), update rule ρ with the replacement rule for τ-projection (line 15), and remove all of the variables we projected so far from V (line 16). *Otherwise*, we remove v from V to eliminate consideration of projecting it again. We repeat this process until set V is empty (lines 4-21).

Recall that ρ_2 is rule (4). To illustrate Algorithm 1, we present the execution of `Projection`$(\rho_2, \beta, 2)$ as a table in Figure 1. The first and second rows of the table state the values of variables $V, \mathbb{R}, \rho$ at line 4 during the first and second iterations of the while-loop respectively. The last row presents the values of these variables at line 22.

It is due to note that procedure `Projection` is nondeterministic due to line 5. In case of our illustration on `Projection`$(\rho_2, \beta, 2)$ there are two possibilities to con-

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

Algorithm 1: *Projection*

Input : ρ: rule of the form $a \leftarrow \mathbb{B}$
τ: projection type α or β
n: positive integer (order of projection)
Output: a pair where the first element is the rewritten rule and the second element is the set of τ rules produced

Function `Projection`(ρ, τ, n)**:**
  $V \leftarrow \{v \mid v$ is a variable in ρ that doesn't occur in its head and $|\alpha(\rho, \{v\}\uparrow\omega)| \leq n\}$;
  $\mathbb{R} \leftarrow \emptyset$;
  while $V \neq \emptyset$ **do**
    $v \leftarrow$ a variable in V such that there is no variable v' in V where $|\alpha(\rho, \{v\}\uparrow\omega)| > |\alpha(\rho, \{v'\}\uparrow\omega)|$;
    $W \leftarrow \{v\}$;
    foreach *w in V different from v* **do**
      if $\alpha(\rho, \{w\}\uparrow\omega) \subseteq \tau(\rho, \{v\}\uparrow\omega)$ **then**
        $W \leftarrow W \cup \{w\}$;
      end
    end
    $s \leftarrow$ the τ rule of τ projecting W from ρ;
    if *the set of variables occurring in the body of s is different from the set of variables in the body of ρ* **then**
      $\mathbb{R} \leftarrow \mathbb{R} \cup \{s\}$;
      $\rho \leftarrow$ the replacement rule of τ projecting process;
      Delete all elements in W from V;
    end
    else
      Delete v from V;
    end
  end
  return $(\rho, \mathbb{R})$;
End Function

While-Loop Iterations	Variables	Values
1	V	$\{B, C, E\}$
	$\mathbb{R}$	$\emptyset$
	ρ	ρ_2
2	V	$\{B, C\}$
	$\mathbb{R}$	$\{s'(D) \leftarrow s(D, E).\}$
	ρ	$p(A, D, F) \leftarrow q(A, B, C), r(B, D), u(C), w(F), s'(D).$
Return	V	$\emptyset$
	$\mathbb{R}$	$\{s'(D) \leftarrow s(D, E).$
		$qr'(A, D) \leftarrow q(A, B, C), r(B, D), u(C), s'(D).\}$
	ρ	$p(A, D, F) \leftarrow w(F), s'(D), qr'(A, D).$

Fig. 1. Algorithm 1 illustration with $\rho = \rho_2$, $\tau = \beta$, and $n = 2$.

sider. In both cases the procedure will return rules of the form

$$s'(D) \leftarrow s(D, E).$$
$$p(A, D, F) \leftarrow w(F), s'(D), proj_{bc}(A, D).$$

whereas it will differ on the other rules presented in the table below.

possibility 1	possibility 2
$proj_c(A, B) \leftarrow q(A, B, C), u(C).$ $proj_{bc}(A, D) \leftarrow proj_c(A, B), r(B, D), s'(D).$	$proj_{bc}(A, D) \leftarrow q(A, B, C), r(B, D),$ $u(C), s'(D).$

Implementation Details System CLINGO version 5.3.0 is an answer set programming tool chain that incorporates grounder GRINGO [13, 17] and solver CLASP [16]. Our implementation of PROJECTOR utilizes PYCLINGO, a sub-system of CLINGO that provides users with various system enhancements through the scripting language Python. One such enhancement allows us to intervene in the workings of CLINGO. The PROJECTOR system uses PYCLINGO to parse a logic program and turn it into a respective abstract syntax tree. At this point, the PROJECTOR subroutines take over by analyzing the parsed program and modifying its "normal" rules according with the described procedures. (By "normal" we refer to the rules of the form discussed in this paper. Yet, the language of CLINGO offers its users more sophisticated constructs in its rule, for example, aggregates. The PROJECTOR ignores such rules.) Once program rewritings are performed on the level of the abstract syntax tree representation of the program the control is given back to PYCLINGO that continues with grounding and then solving. The PYCLINGO interface allows us to guarantee that the system PROJECTOR is applicable to all programs supported by CLINGO version 5.3.0.

System PROJECTOR can be controlled with various flags that determine which type of projection to perform. At the url of the system given at Footnote 4, the flags are described in detail. It is important to note that the flag `--random` allows a user to specify a seed that is used to carry out the nondeterministic decision of line 5 in the `Projection` algorithm.

4 Experimental Analysis

In our experiments we utilize the application called ASPCCG described in [20] and three benchmarks, namely, *Stable Marriage, Permutation Pattern Matching, and Knight Tour with Holes* stemming from the Fifth Answer Set programming Competition [9]. In ASPCCG, the authors formulate the task of parsing natural language, namely, recovering the internal structure of sentences, as a planning problem in answer set programming. The three other mentioned benchmarks were used by the authors of the system LPOPT [3] to report on its performance.

Results on ASPCCG Our choice of the ASPCCG application, is due to the fact that a prior extensive experimental analysis was performed on it in [5]. We now restate some of these earlier findings relevant to our analysis. System ASPCCG version 0.1 (ASPCCG-0.1) and ASPCCG version 0.2 (ASPCCG-0.2) vary only in how specifications of the planning problem are stated, while the constraints of the problem remain the

same. Yet, the performance of ASPCCG-0.1 and ASPCCG-0.2 differs significantly for longer sentences. The way from ASPCCG-0.1 to ASPCCG-0.2 comprised 20 encodings, and along that way, grounding size and solving time were the primary measures directing the changes in the encodings. Rewriting suggestions by Gebser et al. [14] guided the ASPCCG encodings tuning. These suggestions include such "hints on modeling" as

> Keep the grounding compact:
> (i) If possible, use aggregates; (ii) Try to avoid combinatorial blow-up; (iii) Project out unused variables; (iv) But don't remove too many inferences!

In our experiments we consider three encodings out of the mentioned 20:

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

Lierler and Schüller [20] describe the procedure of acquiring instances of the problem using CCGbank[5], a corpus of parsed sentences from real world sources. In [5], the authors report on the performance of answer set solver CLASP v 2.0.2 on a set of 30 randomly selected problem instances from CCGbank that were used in performance tuning by Lierler and Schüller. In Figure 2, we reproduce their findings for ENC1, ENC7, and ENC19. The second column presents the total number of timeouts/memory outs (3000 sec. timeout), the third column presents the average solving time (in seconds; on instances that did not timeout/memoryout), and the last column reports a number n so that n and 10^5 are factors relating to the average number of ground rules reported by CLASP v 2.0.2. These numbers were obtained in experiments using a Xeon X5355 @ 2.66GHz CPU. The presented table illustrates that the selected ENC1, ENC7, and ENC19 encodings differ substantially. It is interesting to note that on the way from ENC1 to ENC7 a software engineer applied projection technique once and from ENC7 to ENC19 three times. In order to conduct extensive analysis on the impact of solver's configuration

Encoding	# timeout/memout	solving in sec	factor w.r.t. grounding
ENC1	6	301	14
ENC7	5	138	4
ENC19	2	128	8

Fig. 2. ASPCCG Performance

on ASPCCG, Buddenhagen and Lierler [5] separated CCGbank instances (sentences) by word count into five word intervals restricting attention to sentences having between 6 and 25 words. They then randomly selected an equal number of sentences from each class when creating different sets of instances. In our experiments, we utilize the set of 60 instances that they call held-out set. We present the results for the hardest 20 instances in this set based on the performance of PYCLINGO (using the default configuration of gringo) on ENC19. In all figures the instances are given on the x-axis sorted by the performance of PYCLINGO on ENC19. We benchmarked α and β projections using the greatest possible order for each case. We also provide the results for the LPOPT

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

system. In the figures, we present data on runtimes (time spent in grounding and solving) and size of ground programs (number of ground rules). In all figures, ENC1, ENC7, and ENC19 present numbers associated with PYCLINGO (using the default configuration of gringo) on the respective encoding. We used an Intel® Core™ i5-4250U CPU @ 1.30GHz CPU.

Figure 3 presents the total runtime in seconds for ENC1 for three distinct variants of β-projection. The numbers 123, 456, and 789 are the seeds passed on to the system with the `--random` flag. We observe that a seed may affect the performance of the system significantly. In the remainder, we present the results for the 123 seed only, to keep the graphs readable. (The choice of this seed is arbitrary.) We use the same seed for presenting the results on α projection. System LPOPT also displays nondeterministic behavior, where the flag `-s` is used to specify a seed. We present the results on LPOPT using the seed 123.

Figure 4 compares the runtime of all considered systems, namely, PYCLINGO, α-PROJECTOR, β-PROJECTOR, and LPOPT. Neither α nor β-projection show any significant performance loss across the experimented instances. In nearly all instances, PROJECTOR outperforms PYCLINGO. Of the 20 displayed instances, 16 show improvement for β-projection over α-projection. Besides the effects of randomness, we attribute the difference in the performance of α and β-projection to the additional literals of β-rule (in comparison to α-rule) that act as "guards"[6]. These guards impose further limits on the domains of variables occurring in them so that a grounding procedure implemented in PYCLINGO benefits from these additional restrictions Figure 5 presents the data on sizes of ground programs for the same systems and instances. For these instances, β-projection results in a reduction of grounding size compared to α-projection for all instances. This can again be attributed to the guards added by β-projection. This is not surprising, as no new variables are introduced into the β-rule when compared to its α-rule counterpart, so the additional guard literals can only restrict the grounding size. It is also obvious that there is a strong correlation between the reduction in grounding size and improvement of runtime.

Figure 6 compares the runtime of all considered systems on the ENC7 encoding. Here we again note that β-projection is generally superior to α-projection, yet we cannot claim that β-projection improves on the original encoding ENC7. Figure 7 illustrates that both α and β-projection produce ground programs that are larger than these produced by PYCLINGO given the original encoding. This once more illustrates the strong correlation between the reduction in grounding size and improvement of runtime.

Figure 8 compares the runtime of all considered systems on ENC19. Figure 9 presents the data on sizes of ground programs. These graphs illustrate similar behavior of the systems as in the case of the ENC7 encoding.

Figure 10 presents the runtimes for PYCLINGO on ENC1, ENC7, and ENC19 together with the runtime of β-PROJECTOR on ENC1. It shows that β-projection can be used to supplement human efforts in performance tuning, as the runtime of β-PROJECTOR on ENC1 is comparable to that of PYCLINGO on ENC7. Recall that ENC7 was obtained from ENC1 using 7 iterations by a human and substantial experimental analysis between these iterations (see [5] for details).

[6] The term *guards* was suggested by Miroslaw Truszczynski.

So far the presented experiments illustrate that the rule decomposition method implemented in LPOPT is superior to both α and β-projection techniques introduced here. The encodings of ASPCCG contain rules with aggregate expressions. System LPOPT is capable to perform its decomposition method also within these expressions, whereas PROJECTOR ignores the aggregates. It is a direction of future research to expand the capabilities of PROJECTOR to handle aggregates. Then, a more fair comparison on the ASPCCG domain can be made between the systems.

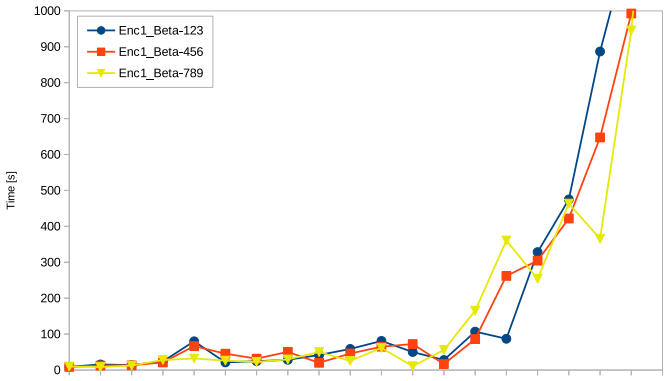

Fig. 3. ENC1: Runtime of β-PROJECTOR with different random seeds

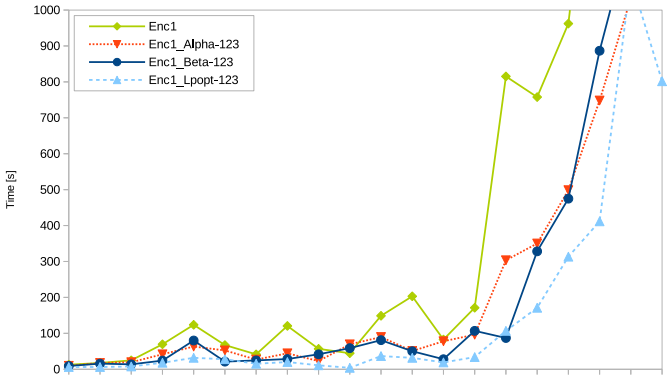

Fig. 4. ENC1: Runtime

Results on Stable Marriage and more Figures 11 and 12 present the runtimes of β-projection and LPOPT with three distinct seeds on Stable Marriage and Permutation Pattern Matching domains. *No aggregates* were used in the benchmarked encodings of these problems. It is apparent that the LPOPT system is truly sensitive to a provided seed. System PROJECTOR exhibits comparable performance between its variants.

In case of Stable Marriage, β-projection always outperforms PYCLINGO on the original encoding. In case of LPOPT, two of its variants exhibit substantially worse behavior than PYCLINGO. In case of Permutation Pattern Matching, it is safe to say that in general program rewriting techniques implemented in LPOPT and PROJECTOR are of benefits.

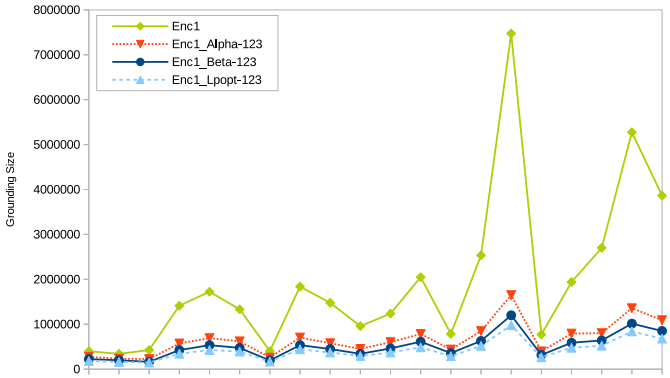

Fig. 5. ENC1: Grounding size

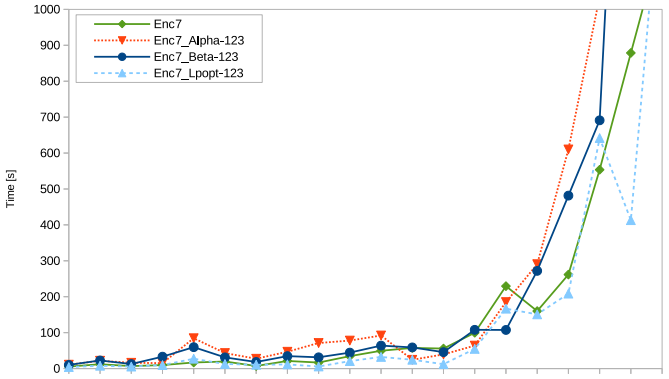

Fig. 6. ENC7: Runtime

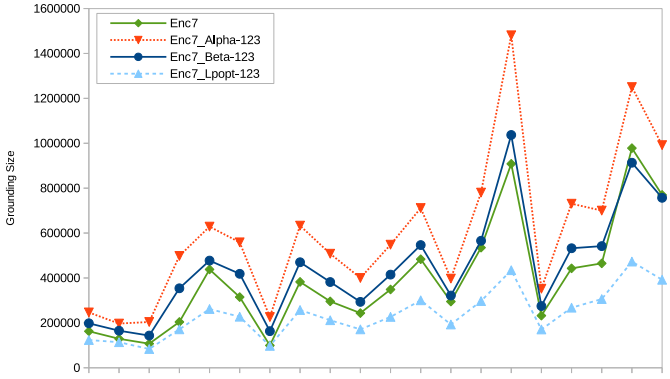

Fig. 7. ENC7: Grounding size

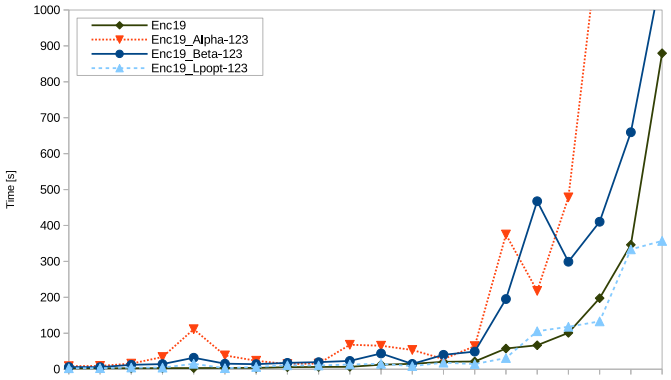

Fig. 8. ENC19: Runtime

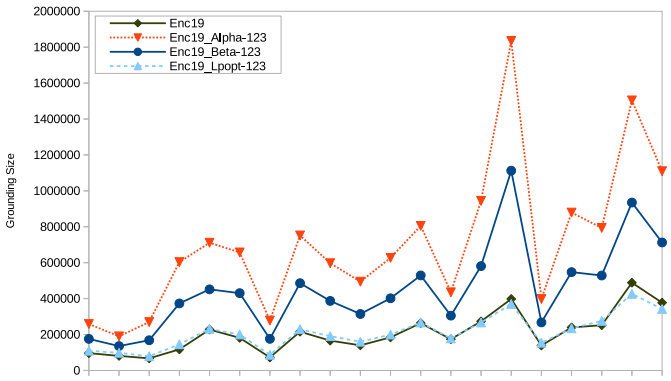

Fig. 9. ENC19: Grounding size

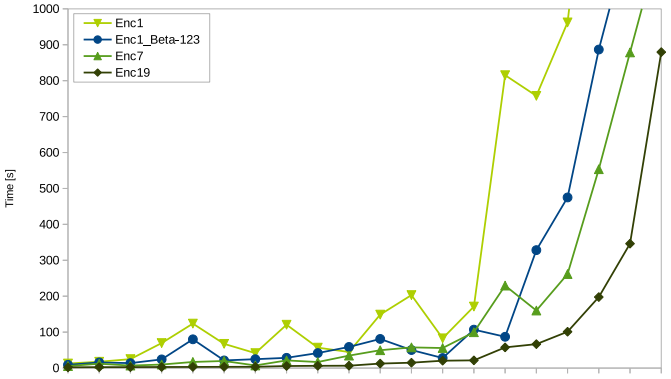

Fig. 10. ENC1,ENC7,ENC19: Runtime

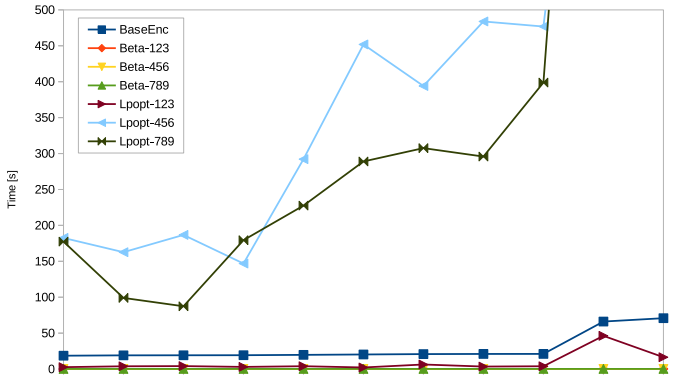

Fig. 11. Stable Marriage: Runtime

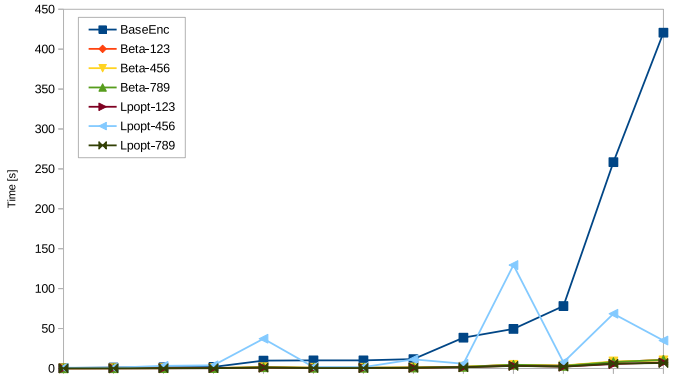

Fig. 12. Permutation Pattern Matching: Runtime

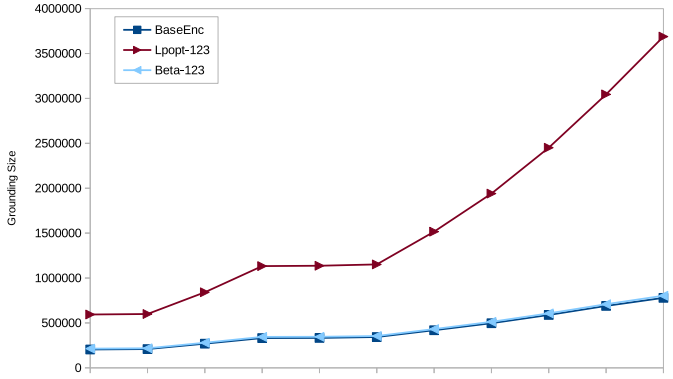

Fig. 13. Knight Tour with Holes: Grounding size

The last benchmark that we consider is Knight Tour with Holes. This benchmark proves too difficult to be solved within 300 sec time limit for any of the considered configurations. Yet, it is interesting to see the effect of LPOPT and PROJECTOR on grounding size. System PROJECTOR has no substantial effect on grounding size of the original program (if to zoom in we would observe a slight increase in grounding size across the board). System LPOPT at least doubles the size of any considered instance. Bichler [2] analyzed the behavior of LPOPT on this domain and came to the conclusion that it is the treatment of safety by LPOPT that translates into such a drastic difference.

5 Discussion, Future Work, Conclusions

In this work we introduce the concepts of α and β-projection and state an algorithm called `Projection` that performs these projections iteratively. We then implement the `Projection` procedure in system PROJECTOR. Our experimental analysis shows that β-projection outperforms α-projection in almost all cases. We also show that LPOPT is generally superior to both forms of projection on the ASPCCG domain. Possibly, this is due to the LPOPT system's rich language support. Yet, in our remaining experiments we demonstrated that PROJECTOR generates more consistent runtimes compared to LPOPT and also outperforms LPOPT. The results collected through our experiments open up several directions of future work:

- *Grounding size prediction and heuristics* As demonstrated in our experimental analysis, there is a strong correlation between grounding size and runtime. It is clear that not all projections result in performance gains. As such, it is reasonable to believe that selectively performing projection according to the resulting predicted grounding size could lead to substantial performance gain. Recent work on LPOPT has shown that a heuristic approach to performing decompositions can lead to performance gains [8]. Similar approaches to PROJECTOR may also be beneficial.
- *Improve language support* Expanding the functionality of PROJECTOR to include language features such as aggregates and optimization statements may enable further performance benefits. This will also allow us to perform a more fair comparison between LPOPT and PROJECTOR on more domains.
- *Data collection* Expanding the experimental analysis to more domains will enable us to better understand the implications behind the use of PROJECTOR.

In conclusion, our experimental analysis shows that system PROJECTOR is a solid step in the direction of providing an automated means for performance tuning in answer set programming.

References

1. Balduccini, M., Gelfond, M., Nogueira, M.: Answer set based design of knowledge systems. Ann. Math. Artif. Intell. **47**(1-2), 183–219 (2006)
2. Bichler, M.: Optimizing non-ground answer set programs via rule decomposition. (2015), Bachelor Thesis, TU Wien
3. Bichler, M., Morak, M., Woltran, S.: lpopt: A rule optimization tool for answer set programming. In: Proceedings of International Symposium on Logic-Based Program Synthesis and Transformation (2016)

4. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM **54**(12), 92–103 (2011)
5. Buddenhagen, M., Lierler, Y.: Performance tuning in answer set programming. In: Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR) (2015)
6. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Computable functions in ASP: theory and implementation. In: Proceedings of International Conference on Logic Programming (ICLP). pp. 407–424 (2008)
7. Calimeri, F., Fusca, D., Perri, S., Zangari, J.: I-dlv: The new intelligent grounder of dlv. Intelligenza Artificiale **11**(1), 5–20 (2017)
8. Calimeri, F., Fusca, D., Perri, S., Zangari, J.: Optimizing answer set computation via heuristic-based decomposition. In: International Symposium on Practical Aspects of Declarative Languages. pp. 135–151. Springer (2018)
9. Calimeri, F., Gebser, M., Maratea, M., Ricca, F.: Design and results of the fifth answer set programming competition. Artificial Intelligence **231**, 151 – 181 (2016). https://doi.org/https://doi.org/10.1016/j.artint.2015.09.008, http://www.sciencedirect.com/science/article/pii/S0004370215001447
10. Eiter, T., Fink, M., Tompits, H., Traxler, P., Woltran, S.: Replacements in non-ground answer-set programming. In: Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR) (2006)
11. Eiter, T., Traxler, P., Woltran, S.: An implementation for recognizing rule replacements in non-ground answer-set programs. In: Proceedings of European Conference On Logics In Artificial Intelligence (JELIA) (2006)
12. Faber, W., Leone, N., Mateis, C., Pfeifer, G.: Using database optimization techniques for nonmonotonic reasoning. pp. 135–139 (1999)
13. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A user's guide to gringo, clasp, clingo, and iclingo. (2010), available at http://potassco.sourceforge.net
14. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Challenges in answer set solving. In: Balduccini, M., Son, T. (eds.) Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays in Honor of Michael Gelfond, vol. 6565, pp. 74–90. Springer (2011)
15. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Advanced preprocessing for answer set solving. In: Proceedings of the 2008 Conference on ECAI 2008: 18th European Conference on Artificial Intelligence. pp. 15–19. IOS Press, Amsterdam, The Netherlands, The Netherlands (2008), http://dl.acm.org/citation.cfm?id=1567281.1567290
16. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artificial Intelligence **187**, 52–89 (2012)
17. Gebser, M., Schaub, T., Thiele, S.: Gringo: A new grounder for answer set programming. In: Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning. pp. 266–271 (2007)
18. Lierler, Y.: SAT-based Answer Set Programming. Ph.D. thesis, University of Texas at Austin (2010)
19. Lierler, Y., Maratea, M., Ricca, F.: Systems, engineering environments, and competitions. AI Magazine **37**(3) (2016)
20. Lierler, Y., Schüller, P.: Parsing combinatory categorial grammar with answer set programming: Preliminary report. In: Correct Reasoning: Essays on Logic-Based AI in Honor of Vladimir Lifschitz. Springer (2012)
21. Ricca, F., Grasso, G., Alviano, M., Manna, M., Lio, V., Iiritano, S., Leone, N.: Team-building with answer set programming in the gioia-tauro seaport. Theory and Practice of Logic Programming **12**(3), 361–381 (2012)