2013

Prolog and ASP Inference Under One Roof

Marcello Balduccini
Eastman Kodak

Yuliya Lierler
University of Nebraska at Omaha, ylierler@unomaha.edu

Peter Schüller
Sabancı University

Prolog and ASP Inference Under One Roof

Marcello Balduccini[1], Yuliya Lierler[2], and Peter Schüller[3]

[1] Eastman Kodak Company, USA
marcello.balduccini@gmail.com
[2] University of Nebraska at Omaha, USA
ylierler@unomaha.edu
[3] Sabancı University, Turkey
peterschueller@sabanciuniv.edu

Abstract. Answer set programming (ASP) is a declarative programming paradigm stemming from logic programming that has been successfully applied in various domains. Despite amazing advancements in ASP solving, many applications still pose a challenge that is commonly referred to as *grounding bottleneck*. Devising, implementing, and evaluating a method that alleviates this problem for certain application domains is the focus of this paper. The proposed method is based on combining backtracking-based search algorithms employed in answer set solvers with SLDNF resolution from PROLOG. Using PROLOG inference on non-ground portions of a given program, both grounding time and the size of the ground program can be substantially reduced.

Keywords: Answer Set Programming, Prolog, Grounding Bottleneck

1 Introduction

Answer set programming (ASP) [4] is a declarative programming paradigm stemming from a knowledge representation and reasoning formalism based on the answer set semantics of logic programs. It can be used whenever we want to solve a search problem where the goal is to find solutions among a finite, but potentially very large, number of possibilities. ASP has been successfully applied in different areas of knowledge representation and computer science, including Space Shuttle control [25] and Linux package configuration [13]. Most modern answer set solving tools encapsulate two systems: a grounder, such as LPARSE or GRINGO, and an answer set solver, such as CMODELS or CLASP. A grounder is a software system that takes a logic program *with* variables as an input and produces an equivalent program *without* variables – a ground program. An answer set solver is then invoked on a ground program to generate its answer sets. Answer set solvers typically rely on the enhancements of the Davis-Putnam-Logemann-Loveland procedure [6] – classic backtracking-based search algorithm. Despite amazing advancements in solving technology, many applications still pose a challenge. *Grounding bottleneck* refers to situations where grounding results in programs that are too large for the solving tools to handle effectively. Alleviating grounding bottleneck is the main focus of this work. We describe, implement, and evaluate an approach for combining backtracking-based search algorithms of answer set solvers with SLDNF resolution

from PROLOG. As a result, the newly implemented approach makes it possible to avoid the grounding of portions of a program by delegating the processing of those parts to a PROLOG system.

The grounding bottleneck has been recognized as a serious issue in recent years. Constraint answer set programming (CASP) [18] is one of the directions of research that has been largely motivated by an attempt to solve the problem. It integrates answer set programming with constraint (logic) programming, which allows applying constraint processing techniques for effective reasoning over non-boolean constructs. CASP introduces a notion of constraint atoms that trigger additional processing by constraint programming tools and at the same time may reduce the size of the grounding. Mellarkod et al. [22] developed one of the earliest CASP languages called *AC*. They also introduced an algorithm for a special class of programs in that language. The primary focus of the work on *AC* was integrating efficient constraint processing capabilities into answer set solving methods. Yet, [22] touched on another crucial aspect of the integration of ASP and CLP: integrating ASP backtracking search with PROLOG SLDNF resolution. In the present paper we resume and expand the investigation on this topic, focusing on a special case of *AC* programs that consist of "standard" (non-constraint) answer set programs.

More on related work: Works by Alviano and Faber [1], de Cat et al. [5], Eiter et al. [8, 7] are other interesting attempts to alleviate grounding issues. Alviano and Faber propose a *magic sets*-based program rewriting method as a query optimization technique in ASP. This method helps an answer set solver prune the search space by disregarding parts of the program irrelevant to a given query. The goal is achieved by rewriting an original program (if a class of a program permits) in a form that guides the computation by the ASP grounder and solver by taking advantage of information provided by the query. The approach attempts to "mimic" PROLOG-like behavior using ASP technology. The approach advocated here is orthogonal. We propose to take advantage of the PROLOG engine itself when possible. Techniques in the spirit of incremental answer set programming [12] were developed by de Cat et al. [5] and employ a "grounding as needed" approach in solving. The DLVHEX solver [8, 7] also provides a possibility for grounding as needed: it uses special *Splitting Sets* to process parts of a program in a sequence, so that the grounding of the *current* part depends on the answer sets of the *previous* parts.

Paper Structure: We start the presentation by a review of preliminary concepts as well as a special case of *AC* programs that are at the center of attention in this work. We then introduce a variant of the *AC* algorithm and describe its implementation within the CASP solver EZCSP [3]. We conclude with a discussion on an experimental analysis that we conducted to assess the introduced technique.

2 Hybrid Programs

A *logic program* is a finite set of rules of the form

$$a_0 \leftarrow a_1, \ldots, a_l, \mathit{not}\ a_{l+1}, \ldots, \mathit{not}\ a_m,\ \mathit{not\ not}\ a_{m+1}, \ldots,\ \mathit{not\ not}\ a_n, \qquad (1)$$

where a_0 is $\bot$ or an atom, and each a_i $(1 \leq i \leq n)$ is an atom. Atoms may be non-ground. We call a rule a *constraint*, if $a_0 = \bot$. This is a special case of programs with nested expressions [21]. We assume that the reader is familiar with the definition of an answer set of such programs and refer to the paper by Lifschitz et al. ([21]) for details. According to [11], a *choice rule* $\{a\}$ of the LPARSE[4] language [23] can be seen as an abbreviation for a rule $a \leftarrow \ not\ not\ a$. We adopt this abbreviation in the rest of the paper.

The expression a_0 is the *head* of rule (1). If B denotes the *body* of (1), the right hand side of the arrow, we write B^{pos} for the elements occurring in the *positive* part of the body, i.e., $B^{pos} = \{a_1, \ldots, a_l\}$.

To process a logic program, or in other words, to find answer sets of a program or establish some properties about its answer sets, such software systems as answer set solvers and sometimes PROLOG interpreters are used. A sample logic program is:

$$\begin{array}{l} down(T) \leftarrow not\ on. \\ down(0).\ down(1).\ \ldots\ down(3600). \\ okTime(T) \leftarrow not\ down(T). \\ \bot \leftarrow occurs(a,5000),\ not\ okTime(5000). \\ occurs(a,5000). \\ \{on\}. \end{array} \tag{2}$$

This program has a unique answer set

$$\{occurs(a,5000),\ okTime(5000),\ on,\ down(0),\ldots,down(3600)\}. \tag{3}$$

Note that neither answer set solvers nor PROLOG systems can handle such a program. First, program (2) contains a constraint and a choice rule, which makes PROLOG systems inapplicable. Second, (2) contains a rule

$$okTime(T) \leftarrow not\ down(T),$$

which violates the common *safety* condition imposed by ASP grounders. A safe rule is such that each variable occurring in its head or its negative part of the body appears in the positive part of the body. Nevertheless, the first three lines of (2) form a logic program that may be processed by PROLOG systems, whereas the last three lines form a program that is acceptable by an answer set solver. In a sense, program (2) is a "hybrid" program that borrows acceptable features from two worlds of logic programming: "classic" PROLOG programming and answer set programming. In this paper we present an algorithm (a family of algorithms) that takes advantage of two inference technologies that are usually used disjointly in logic programming, in PROLOG systems and in answer set solvers. As a result programs such as (2) can be processed by a solver supporting such an algorithm. We implement a variant of this algorithm in the solver EZCSP[5] [3].

In order to treat parts of a program differently (using PROLOG inference in one case, and answer set solver inference in another) we identify a group of program predicates

[4] `http://www.tcs.hut.fi/Software/smodels/`
[5] `http://marcy.cjb.net/ezcsp/`

that we use to guide the splitting of the program into two disjoint parts. To make it precise we introduce the following notation.

For a program Π and a set $\mathbf{p}$ of predicate symbols, the part of Π that consists of all the rules whose heads are atoms formed using predicate symbols from $\mathbf{p}$ is denoted by $\Pi_{\mathbf{p}}$. By $\Pi_{\mathbf{p}}^{-}$ we denote $\Pi \setminus \Pi_{\mathbf{p}}$. For example, let Π stand for (2) and let $\mathbf{p}_1$ be the set of predicate symbols

$$\{okTime,\ down\}. \tag{4}$$

Then, $\Pi_{\mathbf{p}_1}$ is:

$$\begin{array}{l} down(T) \leftarrow not\ on. \\ down(0).\ \ldots\ down(3600). \\ okTime(T) \leftarrow not\ down(T), \end{array} \tag{5}$$

whereas

$$\begin{array}{l} \bot \leftarrow occurs(a,5000),\ not\ okTime(5000) \\ occurs(a,5000). \\ \{on\}. \end{array} \tag{6}$$

is $\Pi_{\mathbf{p}_1}^{-}$. For a program Π, by $ground(\Pi)$ we denote the set of all ground instances of all rules in Π. We say that Π is *semi-ground* w.r.t. a set $\mathbf{p}$ of predicate symbols if $\Pi_{\mathbf{p}}^{-}$ is a ground program (i.e., contains no variables) and $\Pi_{\mathbf{p}}$ is such that all of its non-ground atoms are formed from predicate symbols in $\mathbf{p}$. For example, program (2) is semi-ground w.r.t. predicate symbols (4).

For any atom $p(\mathbf{t})$, by $p(\mathbf{t})^0$ we denote its predicate symbol p. For any program Π, the *predicate dependency graph* of Π is the directed graph that

- has all predicates occurring in Π as its vertexes, and
- for each rule (1) in Π has an edge from a_0^0 to a_i^0 where $1 \leq i \leq l$.

We say that a program Π is *splittable* w.r.t. predicate symbols $\mathbf{p}$ if each strongly connected component of the predicate dependency graph of Π is either a subset of $\mathbf{p}$ or a disjoint set from $\mathbf{p}$. Program (2) is splittable w.r.t. predicate symbols (4).

The hybrid algorithm that we propose in this note is applicable to splittable programs. To present this algorithm we introduce several concepts.

Given a program Π and a set $\mathbf{p}$ of predicate symbols, a set X of atoms is a ***p**-input answer set* (or an input answer set w.r.t. $\mathbf{p}$) of Π if X is an answer set of $\Pi \cup X_{\mathbf{p}}^{-}$ where by $X_{\mathbf{p}}^{-}$ we denote the set of atoms in X whose predicate symbols are different from those occurring in $\mathbf{p}$. [6] For instance, let X be a set $\{a(1), b(1)\}$ of atoms and let $\mathbf{p}$ be a set $\{a\}$ of predicates, then $X_{\mathbf{p}}^{-}$ is $\{b(1)\}$. The set X is a $\mathbf{p}$-input answer set of a program $a(1) \leftarrow b(1)$. On the other hand, it is not an input answer set for the same program with respect to a set $\{a,b\}$.

By $At(\Pi)$ we denote the set of all atoms occurring in a program Π.

Proposition 1. *For a program Π and a set $\mathbf{p}$ of predicate symbols, if Π is splittable then a set of atoms A over $At(ground(\Pi))$ is an answer set of Π iff A is an input answer set of $\Pi_{\mathbf{p}}$ w.r.t. $\mathbf{p}$ and A is an input answer set of $\Pi_{\mathbf{p}}^{-}$ w.r.t. predicate symbols in $\Pi_{\mathbf{p}}^{-}$ different from $\mathbf{p}$.*

[6] Intuitively set $\mathbf{p}$ denotes a set of intentional predicates [10]. The concept of $\mathbf{p}$-input answer sets is closely related to "$\mathbf{p}$-stable models" in [9].

This proposition outlines the basis for our approach. Given a semi-ground and splittable program Π wrt predicate symbols $\mathbf{p}$, we would like to use a PROLOG system for inference over $\Pi_{\mathbf{p}}$ and an answer set solver for inference over $\Pi_{\mathbf{p}}^{-}$. Note that $\Pi_{\mathbf{p}}$ may contain rules that are not ground whereas $\Pi_{\mathbf{p}}^{-}$ is a propositional program so that any answer set solver is applicable to it. Recall that PROLOG is designed to effectively process non-ground programs whereas answer set solvers (without grounders) are able to deal only with propositional programs.

3 Review: Abstract Answer Set Solver

Most state-of-the-art answer set solvers are based on algorithms closely related to the DPLL procedure [6]. Nieuwenhuis et al. described DPLL by means of a transition system that can be viewed as an abstract framework underlying DPLL computation [24]. Our goal is to design a similar framework for describing an algorithm suitable for processing semi-ground splittable programs – QUERY+ASP. As a step in this direction we introduce the graph AS_{Π} that extends the DPLL graph by Nieuwenhuis et al. so that the result can be used to specify an algorithm for finding answer sets of a program.

We frequently identify the body of (1) with the conjunction of its elements (in which *not* is replaced with the classical negation connective $\neg$):

$$a_1 \wedge \cdots \wedge a_l \wedge \neg a_{l+1} \wedge \cdots \wedge \neg a_m \wedge \neg\neg a_{m+1} \wedge \cdots \wedge \neg\neg a_n.$$

Similarly, we often interpret a rule (1) as a clause

$$a_0 \vee \neg a_1 \vee \cdots \vee \neg a_l \vee a_{l+1} \vee \cdots \vee a_m \vee \neg a_{m+1} \vee \cdots \vee \neg a_n \tag{7}$$

(in the case when $a_0 = \bot$ in (1) a_0 is absent in (7)). Given a program Π, we write Π^{cl} for the set of clauses (7) corresponding to all rules in Π.

For a set σ of atoms, a *record* relative to σ is an ordered set M of literals over σ, some possibly annotated by Δ, which marks them as *decision* literals. A *state* relative to σ is a record relative to σ possibly preceding symbol $\bot$. For instance, some states relative to a singleton set $\{a\}$ of atoms are

$$\emptyset,\ a,\ \neg a,\ a^{\Delta},\ a\,\neg a,\ \bot,\ a\bot,\ \neg a\bot,\ a^{\Delta}\bot,\ a\,\neg a\bot.$$

We say that a state is inconsistent if either $\bot$ or two complementary literals occur in it. For example, states $a\,\neg a$ and $a\bot$ are inconsistent. Given a state M, we frequently ignore both annotations and order of elements and consider M as a set of literals possibly including the symbol $\bot$.

If neither a literal l nor its complement occur in M, then l is *unassigned* by M.

If C is a disjunction (conjunction) of literals then by $\overline{C}$ we understand the conjunction (disjunction) of the complements of the literals occurring in C. In some situations, we will identify disjunctions and conjunctions of literals with the sets of these literals.

By $\mathit{Bodies}(\Pi, a)$ we denote the set of the bodies of all rules of a ground program Π with the head a. A set U of atoms occurring in a ground program Π is *unfounded* [26, 16] on a consistent set M of literals with respect to Π if for every $a \in U$ and every

$B \in \mathit{Bodies}(\Pi, a)$, $M \models \overline{B}$ (where B is identified with the conjunction of its elements), or $U \cap B^{pos} \neq \emptyset$.

Each ground program Π determines its *Answer-Set graph* AS_Π. The set of nodes of AS_Π consists of the states relative to the set of atoms occurring in Π. The edges of the graph AS_Π are specified by the transition rules

Unit Propagate: $M \implies M\ l$ if $C \vee l \in \Pi^{cl}$ and $\overline{C} \subseteq M$

Decide: $M \implies M\ l^{\Delta}$ if l is unassigned by M

Fail: $M \implies \bot$ if $\begin{cases} M \text{ is inconsistent and different from } \bot, \text{ and} \\ M \text{ contains no decision literals} \end{cases}$

Backtrack: $P\ l^{\Delta}\ Q \implies P\ \bar{l}$ if $\begin{cases} P\ l^{\Delta}\ Q \text{ is inconsistent, and} \\ Q \text{ contains no decision literals} \end{cases}$

Unfounded: $M \implies M\ \neg a$ if $a \in U$ for a set U unfounded on M wrt Π.

A node is *terminal* in a graph if no edge leaves this node.

For a set M of literals, by $\mathit{pos}(M)$ and $\mathit{neg}(M)$ we denote the set of positive and negative literals in M respectively. For instance, $\mathit{pos}(\{a, \neg b\}) = \{a\}$ and $\mathit{neg}(\{a, \neg b\}) = \{b\}$.

The graph AS_Π can be used for deciding whether a ground program Π has an answer set by constructing a path from $\emptyset$ to a terminal node. The following proposition serves as a proof of correctness and termination for any procedure that is captured by AS_Π.

Proposition 2. *For any ground program Π,*

(a) graph AS_Π is finite and acyclic,
(b) for any terminal state M of AS_Π other than $\bot$, $\mathit{pos}(M)$ is an answer set of Π,
(c) state $\bot$ is reachable from $\emptyset$ in AS_Π if and only if Π has no answer sets.

Let Π be a program (6). The following is a path in AS_Π, with every edge annotated by the name of a transition rule that justifies the presence of this edge in the graph:

$$\begin{array}{l} \emptyset \overset{\textit{Unit Propagate}}{\Longrightarrow} \\ \mathit{occurs}(a, 5000) \overset{\textit{Unit Propagate}}{\Longrightarrow} \\ \mathit{occurs}(a, 5000)\ \mathit{okTime}(5000) \overset{\textit{Unfounded}}{\Longrightarrow} \\ \mathit{occurs}(a, 5000)\ \mathit{okTime}(5000)\ \neg \mathit{okTime}(5000) \overset{\textit{Fail}}{\Longrightarrow} \\ \bot \end{array}$$

Since the last state in the path is terminal and $\bot$, Proposition 3 asserts that this program has no answer sets.

The graph AS_Π is inspired by the graph SM_Π introduced by Lierler [17] for specifying answer set solver SMODELS [23]. The graph SM_Π extends AS_Π by two additional transition rules (in other words, inference rules or propagators): *All Rules Canceled* and *Backchain True*. Lierler and Truszczynski [20] developed a similar framework to model

such modern answer set solvers as CMODELS [15], SUP [17], and CLASP [14]. For the simplicity of this presentation, we settle on the AS_{Π} formalism as a choice for depicting *an* answer set solver. Nevertheless, the procedure described in this paper for combining the inference mechanisms of answer set solving and of PROLOG is not limited to answer set solvers whose algorithm is captured by the AS_{Π} graph. For example, the procedure can be easily adopted by more sophisticated solvers implementing learning, such as CMODELS or CLASP.

4 Abstract QUERY+ASP

Query, Extensions, and Consequences: For a program Π and a set $\mathbf{p}$ of predicate symbols, by $At_{\mathbf{p}}(\Pi)$ we denote a set of atoms occurring in Π whose predicate symbols are in $\mathbf{p}$. By $At^{-}_{\mathbf{p}}(\Pi)$, we denote a set of atoms in Π whose predicates symbols are not in $\mathbf{p}$.

For a semi-ground program Π w.r.t. a set $\mathbf{p}$ of predicate symbols, a (complete) query Q is a (complete) consistent set of literals over $At^{-}_{\mathbf{p}}(\Pi_{\mathbf{p}}) \cup At_{\mathbf{p}}(\Pi^{-}_{\mathbf{p}})$. For a query Q of Π, a complete query E is a *satisfying extension* of Q w.r.t. Π if $Q \subseteq E$ and there is an input answer set A of $\Pi_{\mathbf{p}}$ w.r.t. predicates $\mathbf{p}$ such that $pos(E) \subseteq A$ and $neg(E) \cap A = \emptyset$.

We say that literal l is a consequence of Π and Q if for every satisfying extension E of Q w.r.t. Π, $l \in E$. By $Cons(\Pi, Q)$, we denote the set of all consequences of Π and Q. If there are no satisfying extensions of Q w.r.t. Π we identify $Cons(\Pi, Q)$ with the singleton $\{\bot\}$.

Let Π be (2) and Q be $\{on\}$. The set $\{on, okTime(5000)\}$ forms a satisfying extension of Q w.r.t. Π. Furthermore, this is the only satisfying extension of Q w.r.t. Π. Consequently, it forms $Cons(\Pi, Q)$. On the other hand, there are no satisfying extensions for a query $Q = \{\neg on,\ okTime(5000)\}$ so that $\{\bot\}$ corresponds to $Cons(\Pi, Q)$.

The graph $\text{QAS}_{\Pi,\mathbf{p}}$: For a program Π and a set $\mathbf{p}$ of predicate symbols, by Π^{c} we denote a set of choice rules $\{a\}$ for each atom a in $At_{\mathbf{p}}(\Pi^{-}_{p})$. For instance, let Π be (2) then Π^{c} consists of a choice rule

$$\{okTime(5000)\} \tag{8}$$

Let Π be a logic program and $\mathbf{p}$ a set of predicate symbols. The nodes of the graph $\text{QAS}_{\Pi,\mathbf{p}}$ are the states relative to the set of atoms occurring in Π^{-}_{p}.

The edges of the graph $\text{QAS}_{\Pi,\mathbf{p}}$ include the transition rules of $\text{AS}_{\Pi^{-}_{\mathbf{p}} \cup \Pi^{c}}$. Note how these transition rules take into consideration not only a part of program meant to be processed by an answer set solver $\Pi^{-}_{\mathbf{p}}$ but also its extension with choice rules for atoms whose predicate symbols are in $\mathbf{p}$. For instance, let Π be program (2) and let $\mathbf{p}_1$ be set of predicate symbols (4). The program $\Pi^{-}_{\mathbf{p}_1} \cup \Pi^{c}$ contains the rules of (6) extended with choice rule (8).

Another transition rule that concludes the definition of the graph $\text{QAS}_{\Pi,\mathbf{p}}$ is called *Query Propagate*. To present this rule we introduce the notion of a query. For a state M of $\text{QAS}_{\Pi,\mathbf{p}}$, by $query(M)$ we denote the largest subset of M over $At^{-}_{\mathbf{p}}(\Pi_{\mathbf{p}}) \cup At_{\mathbf{p}}(\Pi^{-}_{\mathbf{p}})$. Let Π be (2) and M be a state $occurs(a, 5000)\ okTime(5000)\ \neg on^{\Delta}$, then $query(M)$ is $\{okTime(5000), \neg on\}$.

The transition rule *Query Propagate* follows

$$\textit{Query Propagate}:\ M \implies M\ l \ \text{ if } \ l \in \mathit{Cons}(\Pi, \mathit{query}(M)).$$

The graph $\text{QAS}_{\Pi,\mathbf{p}}$ can be used for deciding whether a splittable semi-ground program Π w.r.t. predicate symbols $\mathbf{p}$ has an answer set by constructing a path from $\emptyset$ to a terminal node:

Proposition 3. *For any splittable semi-ground program Π w.r.t. predicate symbols $\mathbf{p}$,*

(a) graph $\text{QAS}_{\Pi,\mathbf{p}}$ is finite and acyclic,
(b) for any terminal state M of $\text{QAS}_{\Pi,\mathbf{p}}$ other than $\bot$, $pos(M)$ is a set of all $\Pi^{-}_{\mathbf{p}}$ atoms in some answer set of Π,
(c) state $\bot$ is reachable from $\emptyset$ in $\text{QAS}_{\Pi,\mathbf{p}}$ if and only if Π has no answer sets.

Proposition 3 shows that algorithms, which find a path in the graph $\text{QAS}_{\Pi,\mathbf{p}}$ from $\emptyset$ to a terminal node, can be regarded as solvers for splittable semi-ground programs. We call the class of algorithms captured by the graph QUERY+ASP. Let Π be a program (2). The following is a path in $\text{QAS}_{\Pi,\mathbf{p}}$, with every edge annotated by the name of a transition rule that justifies the presence of this edge in the graph:

$$\begin{array}{l}
\emptyset \overset{\textit{Unit Propagate}}{\Longrightarrow} \\
\mathit{occurs}(a,5000) \overset{\textit{Unit Propagate}}{\Longrightarrow} \\
\mathit{occurs}(a,5000)\ \mathit{okTime}(5000) \overset{\textit{Decide}}{\Longrightarrow} \\
\mathit{occurs}(a,5000)\ \mathit{okTime}(5000)\ \neg \mathit{on}^{\Delta} \overset{\textit{Query Propagate}}{\Longrightarrow} \\
\mathit{occurs}(a,5000)\ \mathit{okTime}(5000)\ \neg \mathit{on}^{\Delta}\ \bot \overset{\textit{Backtrack}}{\Longrightarrow} \\
\mathit{occurs}(a,5000)\ \mathit{okTime}(5000)\ \mathit{on}
\end{array}$$

Since the last state in the path is terminal, Proposition 3 asserts that

$$\{\mathit{occurs}(a,5000),\ \mathit{okTime}(5000),\ \mathit{on}\}$$

is a set of all $\Pi^{-}_{\mathbf{p}}$ atoms in some answer set of Π. Indeed, recall answer set (3).

We note that the $\text{QAS}_{\Pi,\mathbf{p}}$ graph can be seen as a special case of the graph AC_{Π} introduced in [18] for a more sophisticated class of programs called *AC* programs.

5 The "blackbox" QUERY+ASP Algorithm

We can view a path in the graph $\text{QAS}_{\Pi,\mathbf{p}}$ as a description of a process of search for a set of atoms in some answer set of splittable semi-ground program Π by applying the graph's transition rules. Therefore, we can characterize an algorithm of a solver that utilizes the transition rules of $\text{QAS}_{\Pi,\mathbf{p}}$ by describing a strategy for choosing a path in this graph. A strategy can be based, in particular, on assigning priorities to transition rules of $\text{QAS}_{\Pi,\mathbf{p}}$, so that a solver never follows a transition due to a rule in a state if a rule with higher priority is applicable.

The priorities

$$Backtrack, Fail, Unit\ Propagate, Unfounded, Decide >> Query\ Propagate.$$

describe a "*blackbox*" architecture of a QUERY+ASP system that operates as follows: first, it uses an answer set solver on $\Pi^{-}_{\mathbf{p}} \cup \Pi^{c}$ to find an answer set; then it invokes a procedure to verify whether the *Query Propagate* transition is available; if no such transition is available then the answer set found represents a terminal state of $\mathrm{QAS}_{\Pi,\mathbf{p}}$; otherwise, the answer set solver is instructed to look for another answer set and the process is repeated.

PROLOG **for Implementing *Query Propagate***: PROLOG systems can be used to implement the *Query Propagate* transition rule for programs satisfying some additional syntactic constraints. We now discuss one class of such programs.

Let Π be a splittable program w.r.t. predicate symbols $\mathbf{p}$. We say that such a program is PROLOG-*friendly* if $\Pi_{\mathbf{p}}$ is in PROLOG syntax (i.e., contains no rules with nested negation) and acyclic [2, Definition 1.4, Corollary 4.3].[7] Recall that an acyclic program (i) has a unique answer set, and (ii) any PROLOG system terminates on it. Thus a PROLOG system can be used to implement the *Query Propagate* transition rule in a situation in which $query(M)$ assigns all atoms in $At^{-}_{\mathbf{p}}(\Pi_{\mathbf{p}})$. Indeed, PROLOG can be invoked on (i) a program that consists of $\Pi_{\mathbf{p}}$, and atoms (given as facts) occurring positively in $query(M)$ whose predicate symbols are not in $\mathbf{p}$; (ii) a query formed by the literals in $query(M)$ whose predicate symbols are in $\mathbf{p}$.

We refer to the variant of QUERY+ASP that implements the "blackbox" approach and uses PROLOG for *Query Propagate* as PROLOG+ASP. It is a direction of future research to find other means for implementing more general settings of QUERY+ASP.

PROLOG+ASP **implementation in** EZCSP: We expect the reader to be familiar with the syntax of the EZCSP language [3] and with the main principles behind this CASP solver. The EZCSP language has been extended to allow a program Π to contain a declaration, $P(\Pi)$, of the form

$$\#begin\_defined. \quad \Omega \quad \#end\_defined.$$

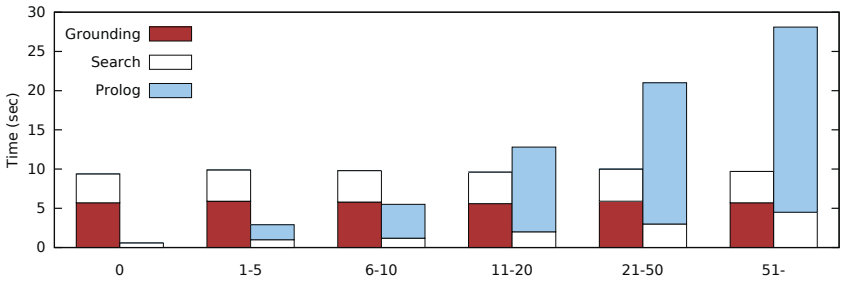
where Ω is an acyclic PROLOG program, which intuitively corresponds to $\Pi_{\mathbf{p}}$. All atoms whose predicate symbols are intended to occur in $\Pi_{\mathbf{p}}$ but not in $\mathbf{p}$ must be prefixed by "*prolog_*" (to notify EZCSP that these atoms are relevant to forming a PROLOG program while implementing *Query Propagate*). All atoms whose predicate symbol is in $\mathbf{p}$ are specified as arguments of the special unary relation "*required*" of the language of EZCSP. For instances, logic program (2) in the modified language of EZCSP is:

$$\begin{array}{l}
\#begin_defined. \\
down(T) \leftarrow not\ prolog_on. \\
down(0).\ down(1).\ \ldots\ down(3600). \\
okTime(T) \leftarrow not\ down(T). \\
\#end_defined. \\
required(okTime(5000)) \leftarrow occurs(a, 5000). \\
occurs(a, 5000) \leftarrow . \\
\{prolog_on\}.
\end{array}$$

[7] More general "PROLOG-friendly" syntactic conditions on programs are possible.

The EZCSP algorithm is extended so that, given an EZCSP program Π, it starts by invoking the answer set solver to compute an answer set A of $\Pi \setminus P(\Pi)$. The PROLOG interpreter is then used to determine if the query formed by the atoms of the form $required(\cdot)$ from answer set A holds for the program consisting of Ω and of the "$prolog\_$"-prefixed atoms from A. If the PROLOG interpreter answers positively, then A is returned. Otherwise, the algorithm iterates, instructing the answer set solver to find another answer set.

6 Experimental Domains and Results



		EE			EE+				
		Ground	Search	Memory	Ground	Search	Prolog	Prolog	Memory
Group	# Instances	sec	sec	MB	sec	sec	sec	# calls	MB
0	11	5.7	3.7	619	0.0	0.6	0.0	0.0	15
1-5	7	5.9	4.0	619	0.0	1.0	1.9	0.4	149
6-10	9	5.8	4.0	620	0.0	1.2	4.3	1.8	150
11-20	13	5.6	4.0	619	0.0	2.0	10.8	5.8	149
21-50	28	5.9	4.1	619	0.0	3.0	18.0	10.1	150
51-	23	5.7	4.0	592	0.0	4.5	23.6	24.3	144
total	91	5.8	4.0	612	0.0	2.6	13.6	10.3	132

Fig. 1. Emergency Exit benchmark results. Instances are grouped by the number of paths from start to goal location. We compare ASP (left stacks) with ASP+Prolog (right stacks) and display the time spent for grounding (dark red), solving (white), and Prolog (light blue) in each stack. Section **EE** in the table shows memory usage, grounding and search time with an encoding in ASP, while **EE+** shows results for PROLOG+ASP and additionally shows the number of calls to PROLOG and the time spent in PROLOG execution.

In this work we designed an experimental domain called *Emergency Exit* to evaluate the implementation of the PROLOG+ASP procedure in EZCSP. Emergency Exit is a planning problem involving a robot on a grid. Some grid cells are occupied by obstacles and cannot be traversed. One unoccupied cell is selected as a goal cell, and another one as an emergency exit. At every time step, the robot can move along the x or y axis by one cell, as long as the destination cell is unoccupied. The goal of the robot is to reach

the goal cell from its initial location in such a way that: (i) doing so takes at most n steps, and (ii) the emergency exit is reachable within k steps from any cell traversed by the robot. We also consider a simpler variant in our analysis that we call *Path Finding*. In this problem the task is to find a path that satisfies the requirement (i). It is easy to see that any solution to the Emergency Exit problem is also a solution to the Path Finding problem but not the other way around.

For our experiments we randomly generated 91 instances with a 100×100 grid, $n = 10$, $k = 194$, cells $(1,6)$, $(6,1)$, and $(100,100)$ marked as a goal, start, and an emergency exit respectively. The instances vary in how obstacles are distributed on a grid: in each case there are between 12 and 25 occupied cells in the part of the grid between $(1,1)$ and $(10,10)$. This randomly varies the number of possible paths from start to goal of length n. Moreover, we selected $k = 194$ and located the emergency exit at $(100,100)$ to ensure that reaching the exit would be possible only for certain paths from start to goal.

In the following presentation, by **PF** we denote an ASP encoding of Path Finding; by **EE** we denote an ASP encoding of Emergency Exit. We constructed **EE** by extending **PF** with an encoding of the reachability requirement (ii). Finally, we constructed variant **EE+** of **EE** in such a way that: (1) the **PF** component is processed by the answer set solver of PROLOG+ASP, whereas (2) **EE+** $\setminus$ **PF** is processed by the PROLOG interpreter used in the implementation of PROLOG+ASP in EZCSP.

The experiments were run on a Linux server with 32 2.4GHz Intel® E5-2665 CPU cores and 64GB memory. Every run used a single core only. As grounder we used GRINGO 3.0.5. To evaluate PROLOG+ASP on **EE+** we used EZCSP 1.6.20b57 with CMODELS 3.85 (running MINISAT v 1.12b) and BPROLOG 7.8 as backends. As a reference we also present the performance of CMODELS 3.85 (running MINISAT v 1.12b) on **EE**. The supporting files can be found at http://www.mbalduccini.tk/ezcsp/lpnmr2013/.

Figure 1 shows the experimental results. We group the instances according to how many answer sets are found by **PF**. This number serves as an upper bound to the number of invocations of the PROLOG interpreter needed in the PROLOG+ASP algorithm to find a solution or establish the unsatisfiability of a problem in the **EE+** encoding. For each instance group, the histogram reports the grounding time at the bottom, followed by the search time, followed by the PROLOG execution time; the left stacks (with dark red) are for **EE**, the right stacks (with light blue) for **EE+**.

First, we observe that **EE** performs nearly the same for all instance groups, including the ratio between grounding (dark red) and solver (white) effort. The grounding size for **EE** is on average 47MB (not shown in the figure).

EE+ performs quite differently. The number of invocations of the PROLOG interpreter by the algorithm greatly affects the efficiency. Groups of instances with up to 10 plans in **PF** can be computed more efficiently with **EE+**, *exhibiting a difference in order(s) of magnitude*. In the instances that require more iterations, the time spent in the PROLOG interpreter dominates the overall time for solving. As PROLOG is never called in group 0, it has particular low memory usage for **EE+**. The time required for grounding **EE+** is nearly zero, and the average size of grounding is 0.3MB, which is much lower than for **EE**. Overall, we observe that for instances where only few or no plans from start to goal exist, **EE+** is significantly faster than **EE**.

7 Conclusions

In this paper we described a method for alleviating the grounding bottleneck by combining backtracking-based search algorithms employed in answer set solvers with SLDNF resolution from PROLOG. By means of experimental evaluations, we have demonstrated that, for problems where constraints have large groundings, using PROLOG as an inference engine over these constraints *may* save grounding time and memory and *may* lead to significant gains in the performance. However this is only true when the part of a program evaluated by an answer set solver of PROLOG+ASP is such that it produces only few candidates that have to be verified against the constraints evaluated by PROLOG. This conclusion aligns well with an observation reported in [19], where a study was conducted, comparing the solving technology of answer set solvers and of constraint answer set solvers. As in PROLOG+ASP, the answer set solving component of a constraint answer set solver has access only to a portion of all the constraints of the problem. The other constraints are processed separately by a constraint solver. Such separation of concerns may be very fruitful in solving the grounding bottleneck, yet it has to be used with care in order not to undermine the advanced technology of answer set solvers.

Acknowledgments

We are grateful to Yuanlin Zhang, Michael Gelfond, Vladimir Lifschitz, and Mirosław Truszczyński for useful discussions related to the topic of this work. Peter Schüller is supported by TUBITAK 2216 Research Fellowship.

References

1. Alviano, M., Faber, W.: Dynamic magic sets and super-coherent answer set programs. AI Commun. 24(2), 125–145 (2011)
2. Apt, K., Bezem, M.: Acyclic programs. New Generation Computing 9, 335–363 (1991)
3. Balduccini, M.: Representing constraint satisfaction problems in answer set programming. In: Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP) (2009)
4. Brewka, G., Niemelä, I., Truszczyński, M.: Answer set programming at a glance. Communications of the ACM 54(12), 92–103 (2011)
5. de Cat, B., Denecker, M., Stuckey, P.J.: Lazy model expansion by incremental grounding. In: Technical Communications of the International Conference on Logic Programming (ICLP). pp. 201–211 (2012)
6. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. Communications of the ACM 5(7), 394–397 (1962)
7. Eiter, T., Fink, M., Ianni, G., Krennwallner, T., Schüller, P.: Pushing efficient evaluation of HEX programs by modular decomposition. In: Logic Programming and Nonmonotonic Reasoning (LPNMR). pp. 93–106 (2011)
8. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: dlvhex: A Prover for Semantic-Web Reasoning under the Answer-Set Semantics. In: Workshop on Applications of Logic Programming in the Semantic Web and Semantic Web Services (ALPSWS). pp. 33–39. CEUR WS (2006)

9. Ferraris, P., Lee, J., Lifschitz, V.: Stable models and circumscription. Artificial Intelligence 175, 236–263 (2011)
10. Ferraris, P., Lee, J., Lifschitz, V., Palla, R.: Symmetric splitting in the general theory of stable models. In: International Joint Conference on Artificial Intelligence (IJCAI). pp. 797–803 (2009)
11. Ferraris, P., Lifschitz, V.: Weight constraints as nested expressions. Theory and Practice of Logic Programming 5, 45–74 (2005)
12. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: Engineering an incremental ASP solver. In: International Conference on Logic Programming (ICLP). pp. 190–205 (2008)
13. Gebser, M., Kaminski, R., Schaub, T.: aspcud: A linux package configuration tool based on answer set programming. In: International Workshop on Logics for Component Configuration (LoCoCo) (2011)
14. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: International Joint Conference on Artificial Intelligence (IJCAI). pp. 386–392. MIT Press (2007)
15. Giunchiglia, E., Lierler, Y., Maratea, M.: Answer set programming based on propositional satisfiability. Journal of Automated Reasoning 36, 345–377 (2006)
16. Lee, J.: A model-theoretic counterpart of loop formulas. In: International Joint Conference on Artificial Intelligence (IJCAI). pp. 503–508. Professional Book Center (2005)
17. Lierler, Y.: Abstract answer set solvers. In: International Conference on Logic Programming (ICLP). pp. 377–391. Springer (2008)
18. Lierler, Y.: On the relation of constraint answer set programming languages and algorithms. In: AAAI Conference on Artificial Intelligence (AAAI). MIT Press (2012)
19. Lierler, Y., Smith, S., Truszczyński, M., Westlund, A.: Weighted-sequence problem: ASP vs CASP and declarative vs problem oriented solving. In: International Symposium on Practical Aspects of Declarative Languages (PADL) (2012)
20. Lierler, Y., Truszczyński, M.: Transition systems for model generators — a unifying approach. Theory and Practice of Logic Programming, International Conference on Logic Programming (ICLP) Special Issue 11(4-5) (2011)
21. Lifschitz, V., Tang, L.R., Turner, H.: Nested expressions in logic programs. Annals of Mathematics and Artificial Intelligence 25, 369–389 (1999)
22. Mellarkod, V.S., Gelfond, M., Zhang, Y.: Integrating answer set programming and constraint logic programming. Annals of Mathematics and Artificial Intelligence (2008)
23. Niemelä, I., Simons, P.: Extending the Smodels system with cardinality and weight constraints. In: Minker, J. (ed.) Logic-Based Artificial Intelligence, pp. 491–521. Kluwer (2000)
24. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). Journal of the ACM 53(6), 937–977 (2006)
25. Nogueira, M., Balduccini, M., Gelfond, M., Watson, R., Barry, M.: An A-Prolog decision support system for the Space Shuttle. In: International Symposium on Practical Aspects of Declarative Languages (PADL). pp. 169–183 (2001)
26. Van Gelder, A., Ross, K., Schlipf, J.: The well-founded semantics for general logic programs. Journal of ACM 38(3), 620–650 (1991)