Space Efficient Evaluation of ASP Programs with Bounded Predicate Arities*

Thomas Eiter
Institute of Information Systems
Vienna University of Technology
Favoritenstraße 9-11, Vienna, Austria
eiter@kr.tuwien.ac.at

Wolfgang Faber
Department of Mathematics
University of Calabria
Via Bucci, Rende, Italy
faber@mat.unical.it

Mushthofa Mushthofa
Institute of Information Systems
Vienna University of Technology
Favoritenstraße 9-11, Vienna, Austria
unintendedchoice@gmail.com

Abstract

Answer Set Programming (ASP) has been deployed in many applications, thanks to the availability of efficient solvers. Most programs encountered in practice have an important property: Their predicate arities are bounded by a constant, and in this case it is known that the relevant computations can be done using polynomial space. However, all competitive ASP systems rely on grounding, due to which they may use exponential space for these programs. We present three evaluation methods that respect the polynomial space bound and a generic framework architecture for realization. Experimental results for a prototype implementation indicate that the methods are effective. They show not only benign space consumption, but interestingly also good runtime compared to some state of the art ASP solvers.

Introduction

In the recent years, Answer Set Programming (ASP) has gained momentum as a novel paradigm for declarative problem solving. Its basic idea, to encode a problem into a nonmonotonic logic program, and to extract its solutions from the answer sets (i.e., models) of this program (similar as in SAT solving), has been successfully deployed to many areas of applications. This is also because a number of efficient ASP solvers are available (e.g., smodels, DLV, or clasp).

For example, given a graph G, we can determine all cliques in G of size $k \geq 2$ using the following simple ASP program. The edges (u, v) are encoded via facts $edge(u, v)$; the following rules determine relevant nodes and guess a clique:

$$node(X) \leftarrow edge(X, \_)$$
$$node(X) \leftarrow edge(\_, X)$$
$$in(X) \vee out(X) \leftarrow node(X)$$

This guess is checked with three rules, where $1 \leq i < j \leq k$:

$$\leftarrow in(X_1), in(X_2), X_1 \neq X_2, \quad not\ edge(X_1, X_2), not\ edge(X_2, X_1)$$
$$ok \leftarrow node(X_1), \ldots, node(X_k), in(X_1), \ldots, in(X_k), \quad X_1 \neq X_2, \ldots, X_i \neq X_j, \ldots X_{k-1} \neq X_k$$
$$\leftarrow not\ ok$$

(here rules with empty head act as constraints). The answer sets of this program yield exactly the cliques of size $\geq k$.

However, when we evaluate this program on a state-of-the-art ASP solver, the space consumption increases rapidly with k (that is, the number of distinct variables in the rules). The reason is a *grounding bottleneck* identified in (Eiter et al. 2007): the solvers first construct an optimized subset of the grounding of the input program (using explicit grounders, like Lparse or GrinGo, or implicit ones like DLV); disjunction (as above) or cyclic negation may prevent that this ground program is small, and in fact it can be *exponential*.

On the other hand, as follows from results in (Eiter et al. 2007), for the above program the space explosion for growing k can be avoided. This is because for programs in which the predicate arities are bounded by a constant, deciding if some answer set exists is in the polynomial hierarchy, thus feasible in *polynomial space* (as is computing an answer set).

Eiter et al. (2007) sketched a method based on meta-interpretation to evaluate programs with bounded predicate arities in polynomial space; while intuitive, we have found that in practice their approach, described in more detail in (Mushthofa 2009), is very inefficient.

This motivates the search for alternative methods, and in particular for ones that may help to improve current ASP solvers. We tackle this and make the following contributions.

- We present three different methods for evaluating logic programs with bounded predicate arities which work within polynomial space. The first two methods handle *head-cycle-free* (HCF) programs (Ben-Eliyahu and Dechter 1994), while the third handles all disjunctive programs.
- We then provide a generic framework architecture for implementing the methods. It builds on an abstract decomposition of the three methods into subtasks, some of which can be handled by current ASP solvers or Prolog systems without incurring exponential space. As a show case, we implemented

*This research has been partly supported by the Austrian Science Fund (FWF) project P20840, by Regione Calabria and EU under POR Calabria FESR 2007-2013 within the PIA project of DLVSYSTEM s.r.l., and by MIUR under the PRIN project LoDeN.

BPA, which is a prototype that instantiates the framework architecture using DLV and XSB as external solvers.

• Finally, we report the results of a benchmark experiment, comparing the space and time usage of BPA against the ASP solvers DLV and GrinGo/claspD on ASP programs with bounded predicate arities (considering answer set existence). Rather than purely random programs, we consider here parameterized programs like the one for clique above which naturally increase the number of variables in rules. Results from the experiment confirm the expectation that the three methods are able to evaluate the test instances in polynomial space, while DLV and GrinGo/claspD show exponential space behavior. Interestingly, on many of the instances, BPA also runs faster than the other two systems.

Our results are rather unexpected, as in general, better space behavior means worse run time. They show the potential of the three methods to improve the efficiency of existing ASP systems, and to overcome the grounding bottleneck.

Proofs of the results are available in (Mushthofa 2009).

Preliminaries

We review some basic concepts in Answer Set Programming.

Syntax A rule r is an expression of the form:

$$a_1 \vee \ldots \vee a_k \leftarrow b_1 \ldots, b_m, \mathit{not}\ b_{m+1}, \ldots, \mathit{not}\ b_n \quad (1)$$

where $k \geq 0$, $n \geq m \geq 0$ and all a_i,b_j are function-free first-order atoms. The set $H(r) = \{a_1, \ldots, a_k\}$ is the *head* and $B(r) = B^+(r) \cup B^-(r)$ the *body* of r, where $B^+(r) = \{b_1, \ldots, b_m\}$ is the *positive body*, and $B^-(r) = \{\mathit{not}\ b_{m+1}, \ldots, \mathit{not}\ b_n\}$ the *negative body*. Rule r is a *fact*, if $n = 0$; *normal*, if $k = 1$; *positive*, if $m = n$; and *definite Horn*, if $k = 1 \wedge n = m$. Sometimes we denote r by $H(r) \leftarrow B(r)$; moreover, we use $\mathit{not}(a) = \mathit{not}\ a$ and $\mathit{not}(\mathit{not}\ a) = a$ for an atom a, and $\mathit{not}(L) = \{\mathit{not}(l) \mid l \in L\}$ for sets of literals.

An answer set program (simply, a *program*) $\mathcal{P}$ is a finite set of rules of the form (1); it is *normal*, *positive* etc. if each rule $r \in \mathcal{P}$ is *normal*, *positive* etc. Throughout this paper, we assume that $\mathcal{P}$ is *safe*, i.e., each rule $r \in \mathcal{P}$ is *safe*, which means that each variable X occurring in r also occurs in $B^+(r)$. By $F(\mathcal{P})$ we denote the set of all facts in $\mathcal{P}$.

Answer Sets Let $HB_\mathcal{P}$ be the Herbrand base of $\mathcal{P}$, i.e., the set of all atoms with predicates and constants from $\mathcal{P}$.

An *interpretation* w.r.t. $\mathcal{P}$ is any subset of $I \subseteq HB_\mathcal{P}$; I *satisfies* a ground rule r, denoted $I \models r$, iff $H(r) \cap I \neq \emptyset$ whenever $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$ hold; I satisfies a program $\mathcal{P}$ (is a *model* of $\mathcal{P}$, written $I \models \mathcal{P}$) iff I satisfies all rules in $grnd(\mathcal{P})$, where $grnd(\mathcal{P})$ is the set of all instances of rules in $\mathcal{P}$ with constants from $\mathcal{P}$.

An *answer set* of a positive program $\mathcal{P}$ is any minimal model of $\mathcal{P}$ w.r.t. set inclusion, The answer sets of an arbitrary program $\mathcal{P}$ are the answer sets of the *GL-reduct* (Gelfond and Lifschitz 1988) $\mathcal{P}^I$ of $\mathcal{P}$ w.r.t. I, which is the positive program obtained by deleting from $grnd(\mathcal{P})$ all rules r such that $B^-(r) \cap I \neq \emptyset$ and $B^-(r)$ from all other rules. The set of all answer sets of $\mathcal{P}$ is denoted by $AS(\mathcal{P})$.

The program $\mathcal{P}$ is said to be *consistent* if it has at least one answer set (i.e., $AS(\mathcal{P}) \neq \emptyset$), and *inconsistent* otherwise.

The *positive dependency graph* of a program $\mathcal{P}$ is a directed graph $G^+(\mathcal{P})$ whose nodes are the predicates occurring in $\mathcal{P}$ and with all edges $p_1 \rightarrow p_2$ such that p_1 occurs in $H(r)$ and p_2 in $B^+(r)$ for some rule $r \in \mathcal{P}$.

A program $\mathcal{P}$ is *head-cycle-free* (HCF) (Ben-Eliyahu and Dechter 1994), iff $G^+(\mathcal{P})$ has no cycle containing two distinct atoms from $h(r)$ for some $r \in grnd(\mathcal{P})$. HCF programs can be efficiently rewritten to normal programs using shifting; for any rule r, let $shift(r) = \{a_i \leftarrow B(r), \mathit{not}(H(r) \setminus \{a_i\}) \mid a_i \in H(r)\}$, and for any program $\mathcal{P}$, $shift(\mathcal{P}) = \bigcup_{r \in \mathcal{P}} shift(r)$. It is well-known (Ben-Eliyahu and Dechter 1994) that for every HCF program $\mathcal{P}$, $AS(shift(\mathcal{P})) = AS(\mathcal{P})$; this equality does not hold for arbitrary programs.

Evaluation Methods

We propose three different methods for evaluating a program $\mathcal{P}$ with bounded predicate arities, which stay in polynomial space. The first two methods assume that $\mathcal{P}$ is normal or HCF disjunctive (using $shift(\mathcal{P})$). For handling non-HCF programs we propose a third method.

We begin by defining a positive version of a program, which also drops all constraints. For a rule r, let $pos(r) = H(r) \leftarrow B^+(r)$. For a program $\mathcal{P}$, $pos(\mathcal{P}) = \{pos(r) \mid r \in \mathcal{P}, H(r) \neq \emptyset\}$. $pos(shift(\mathcal{P}))$ is definite Horn and has a single answer set $S_\mathcal{P}$. It is not hard to see that every answer set of $\mathcal{P}$ is a subset of $S_\mathcal{P}$.

For a program $\mathcal{P}$, let $Def(\mathcal{P})$ be the set of all definite Horn rules in $\mathcal{P}$ and let $D_\mathcal{P}$ be the single answer set of $Def(\mathcal{P})$. Here, every answer set is a superset of $D_\mathcal{P}$.

Lemma 1 *Every $A \in AS(\mathcal{P})$ satisfies $D_\mathcal{P} \subseteq A \subseteq S_\mathcal{P}$.*

HCF programs

In this section, we assume that the input program $\mathcal{P}$ is HCF and has been transformed into an equivalent normal program by shifting. The following lemma is crucial.

Lemma 2 (Locality lemma) *Let $\mathcal{P}$ be any logic program and $\mathcal{P}' \subseteq \mathcal{P}$. An answer set A of $\mathcal{P}'$ is also an answer set of $\mathcal{P}$ if and only if for each rule $r \in \mathcal{P}$ it holds that $A \models r$.*

This lemma suggests the approach we take for computing an answer set of a HCF logic program: compute "local answer sets", i.e., the answer sets of certain subsets of the program, and then check whether each of the answer set is indeed a "global answer set", i.e., answer set of the original program, by checking whether it satisfies the program. We perform this check by forming its constraint version $cons(\mathcal{P}) = \{ \leftarrow B(r), \mathit{not}(H(r)) \mid r \in \mathcal{P}\}$, for which the following proposition is easily shown.

Proposition 3 *For an interpretation I and program $\mathcal{P}$, it holds that $I \models \mathcal{P}$ iff $cons(\mathcal{P}) \cup I$ is consistent.*

The methods use certain subsets of a ground program that include exactly one defining rule for each atom in a given set.

Definition 1 *Given a program $\mathcal{P}$ and an interpretation A, $\mathcal{R} \subseteq grnd(\mathcal{P})$ is a* single rule definition program *of $\mathcal{P}$ w.r.t. A iff (i) $F(\mathcal{P}) \subseteq \mathcal{R}$, (ii) for every $a \in A$, there is exactly one rule $r \in \mathcal{R}$ such that $H(r) = \{a\}$, and (iii) for each rule $r \in \mathcal{R}$, $H(r) \subseteq A$, $H(r) \neq \emptyset$, and $B^+(r) \subseteq A$.*

Method 1. Assume $\mathcal{P}$ is a normal program. The main idea is to compute local answer sets by considering smallest sized subsets of $grnd(\mathcal{P})$ which might produce an answer set A of $\mathcal{P}$. As any such A satisfies $A \subseteq S_\mathcal{P}$, intuitively we can obtain A by considering subsets of $grnd(\mathcal{P})$ such that the set of head atoms in the rules yield $S_\mathcal{P}$, which is a single rule definition program of $\mathcal{P}$ w.r.t. $S_\mathcal{P}$, which we call *minimal guess* of $\mathcal{P}$, $R_\mathcal{P}$. Clearly, as $|R_\mathcal{P}| = |S_\mathcal{P}|$ and $S_\mathcal{P}$ is polynomially bounded, we can conclude that every minimal guess of $\mathcal{P}$ is also polynomially bounded. We obtain the following result.

Proposition 4 *Let $\mathcal{P}$ be a normal program. Then for every $A \in AS(\mathcal{P})$ there exists some minimal guess $R_\mathcal{P}$ of $\mathcal{P}$ such that $A \in AS(R_\mathcal{P})$.*

Method 2. The main idea of this method is as follows. As any (local) answer set A of $\mathcal{P}$ satisfies $D_\mathcal{P} \subseteq A \subseteq S_\mathcal{P}$ (cf. Lemma 1), we may find A by iterating over the interpretations A satisfying that condition, and checking for minimality. To prove minimality of A, we need to find a set of ground rules $\mathcal{P}'$ such that A is an answer set of $\mathcal{P}'$. To this end, we define a single rule program $R_{\mathcal{P},A}$ of $\mathcal{P}$ w.r.t. A to be a *supporting minimal guess* of $\mathcal{P}$ w.r.t. A iff for each rule $r \in R_{\mathcal{P},A}$, $\{n \mid \mathit{not}\ n \in B^-(r)\} \cap A = \emptyset$. Since $|R_{\mathcal{P},A}| = |A|$, $A \subseteq S_\mathcal{P}$ and $S_\mathcal{P}$ is polynomially bounded if $\mathcal{P}$ has bounded predicate arities, we conclude that the supporting minimal guesses of $\mathcal{P}$ for any A are also polynomially bounded.

If no supporting minimal guesses of $\mathcal{P}$ w.r.t. A exist, we can safely conclude that A is not an answer set of $\mathcal{P}$. If a supporting minimal guess $R_{\mathcal{P},A}$ of $\mathcal{P}$ w.r.t. A exists, the minimality of A can be proved by showing that A is an answer set of $R_{\mathcal{P},A}$. To this end, we us the following concepts.

Definition 2 *For an atom $a = p(t_1, \ldots, t_k)$, let $\pi(a) = p'(t_1, \ldots, t_k)$, where p' is a new predicate symbol. We extend this function to literals as $\pi(\mathit{not}\ a) = \mathit{not}\ \pi(a)$, to sets of literals as $\pi(S) = \{\pi(a) \mid a \in S\}$, to rules as $\pi(r) = H(r) \leftarrow B^+(r), \pi(B^-(r))$ and to programs as $\pi(R) = \{\pi(r) \mid r \in R\}$. Furthermore, for a set of atoms S, let $\tau(S) = \{\leftarrow \mathit{not}\ a, \pi(a) \mid a \in S\}$.*

We can show the following result:

Lemma 5 *$\pi(R_{\mathcal{P},A}) \cup \pi(A) \cup \tau(A)$ is consistent iff A is an answer set of $R_{\mathcal{P},A}$.*

General programs

For non-HCF disjunctive programs, the property that each answer set A of a program $\mathcal{P}$ is computable from a subset of $grnd(\mathcal{P})$ that has size $|A|$ no longer holds. E.g., for $P = \{a \leftarrow b;\ b \leftarrow a;\ b \vee a\}$, no subset yields the answer set $\{a, b\}$. Thus, the strategy above (using one rule per atom) is not applicable, and it is *a priori* unclear how many rules are needed. In fact, the complexity results in (Eiter et al. 2007) imply that the number can be exponential (unless the polynomial hierarchy collapses to Σ_2^p, which is not expected).

To evaluate non-HCF programs, the approach we use proceeds in two main steps: (i) Generate *sufficiently* many candidate models of $\mathcal{P}$; by Lemma 1, we may use models I of $\mathcal{P}$ such that $D_\mathcal{P} \subseteq I \subseteq S_\mathcal{P}$. (ii) For each candidate I of $\mathcal{P}$ of step (i), we check whether I is the minimal model of the GL-reduct $\mathcal{P}^I$; the answer sets of $\mathcal{P}$ are those I for which the check succeeds. We next discuss these two steps in detail.

Generating models Consider the set $PT_\mathcal{P} = S_\mathcal{P} \setminus D_\mathcal{P}$. We can generate models I of $\mathcal{P}$ satisfying $D_\mathcal{P} \subseteq I \subseteq S_\mathcal{P}$, using the following program (where a' is a fresh atom):

$$G = \{a \vee a' \mid a \in PT_\mathcal{P}\} \cup D_\mathcal{P} \cup cons(\mathcal{P}).$$

Intuitively, $\{a \vee a' \mid a \in PT_\mathcal{P}\} \cup D_\mathcal{P}$ represents a "guess" of all interpretations I such that $D_\mathcal{P} \subseteq I \subseteq S_\mathcal{P}$, while $cons(\mathcal{P})$ enforces that I satisfies $\mathcal{P}$ (cf. Proposition 3).

However, since G contains atoms having disjunctive definitions, evaluating such a program using current ASP solvers (like DLV) may consume exponential space. To avoid this problem, we split $\mathcal{P}$ into $Small(\mathcal{P}) = \{r \in \mathcal{P} \mid |grnd(r)| \leq B(P)\}$, where $B(P)$ is a polynomial bound w.r.t. the size of P (e.g., that rules have at most k variables) and $Big(\mathcal{P}) = \mathcal{P} \setminus Small(\mathcal{P})$.

We perform a two-step computation: (1) generate *candidate models* with guessing rules as in G, but using only $cons(Small(\mathcal{P}))$ instead of $cons(\mathcal{P})$; (2) check these candidate models using the rules in $Big(\mathcal{P})$. Formally, if $I = A \cap S_\mathcal{P}$ for an answer set A of $G_\mathcal{P} = \{a \vee a' \mid a \in PT_\mathcal{P}\} \cup D_\mathcal{P} \cup cons(Small(\mathcal{P}))$, then $D_\mathcal{P} \subseteq I \subseteq S_\mathcal{P}$ and $I \models Small(\mathcal{P})$. If the program $I \cup cons(Big(\mathcal{P}))$ is consistent, I must satisfy $I \models Small(\mathcal{P}) \cup Big(\mathcal{P}) = \mathcal{P}$.

Minimality checking To show that I is a minimal model of $\mathcal{P}^I$ (hence an answer set), we can proceed by enumerating the proper subsets I' of I and ensure that no I' satisfies $\mathcal{P}^I$. The following proposition shows the strategy to perform this.

Proposition 6 *$cons(shift(\pi(\mathcal{P})) \cup \pi(I) \cup I'$ is consistent iff I' satisfies $\mathcal{P}^I$.*

Evaluation Framework

In this section, we propose a framework for efficiently evaluating a program with bounded predicate arities using the three methods proposed in the previous section. The proposed methods involve subtasks, such as checking the consistency of a program, which can be efficiently (i.e., without requiring exponential space) performed using current ASP solvers (see (Drescher et al. 2008; Leone et al. 2006; Simons, Niemelä, and Soininen 2002) for examples).

The following main goals of the framework architecture are (1) to decompose the evaluation of a logic program with bounded predicate arities into simpler subtasks, and (2) to allow the use of current ASP solvers for efficiently performing these subtasks. Figure 1 shows an overview of the framework architecture and the relationship between its components. The main components are the **Controller**, which coordinates the evaluation process performed by the other components and managing input and output operations, and the **MethodSelector**, analyzing the input program to determine (using a heuristics) which method to apply for HCF programs.

Each of the three evaluation methods is implemented in the framework components **EvalMethod1**, **EvalMethod2**, and **EvalDisjunctive**. Each of the methods is then decomposed into subtasks (some of them re-used). **SubSetGen** generates subsets of the ground input program (without generating the full ground program), from which minimal guesses

are created in Method 1 and supporting minimal guesses in Method 2. **ModelGen** generates models A of a program $\mathcal{P}$ as answer set candidates, by iterating over the interpretations satisfying the condition in Lemma 1 and checking whether they satisfy $A \models \mathcal{P}$. These models are used in Method 2 to generate candidate answer sets and in the method for general programs to generate the answer set candidates.

The **ModelChecker** determines whether $I \models \mathcal{P}$. This is required in Method 1 to filter out the local answer sets which are not global answer sets (cf. Lemma 2). The **MinChecker** checks the minimality of a model I for a program $\mathcal{P}$ using Proposition 6. The **ASVerifier** decides whether an interpretation A is an answer set of $\mathcal{P}$ exploiting Lemma 5.

These components can be implemented as series of computations using either a Prolog engine or an ASP solver. In particular, generating subsets of the ground programs in the component **SubsetGen** can be efficiently performed using a Prolog engine, using a simple program rewriting technique, while **ModelGen**, **ModelCheck**, **MinCheck** and **ASVerifier** are encodable as logic programs that can be efficiently evaluated in polynomial space using existing ASP solvers. Thus, we have two additional components: a **PrologEngine** and an **ASPSolver**, which represent the abstraction of the functionalities provided by an external Prolog engine, respectively external ASP solver.

Implementation We have developed a prototype implementation, called BPA, of the framework architecture, using DLV (Leone et al. 2006) and XSB (Sagonas, Swift, and Warren 1994) as external systems. BPA is written in C++ and uses object-oriented data structures and design patterns reusing some data structures of DLVHEX (Schindlauer 2006).

I/O between BPA and DLV uses UNIX's *pipe* mechanism, for reading answer sets in a streaming mode. This is needed to guarantee a polynomial space limit, as programs may have exponentially many answer sets. I/O between BPA and XSB uses XSB's own C language library, which allows to create XSB Prolog *threads* and execute Prolog queries on them.

For efficient evaluation, BPA exploits modular program decomposition using strongly connected components (SCCs) of the program dependency graph similar as in (Eiter, Gottlob, and Mannila 1997; Oikarinen and Janhunen 2008). To avoid storing the (possibly exponentially many) answer sets of the components, they are evaluated using backtracking. Details of the implementation are given in (Mushthofa 2009), the source code is available at http://code.google.com/p/asbpa/.

Experiments

To test how BPA compares against current ASP solvers, we performed experiments running BPA, DLV and claspD,[1] measuring the time and space consumption of the systems.

Benchmark suite We considered six benchmark problems of different nature, and report three representative here.

1. 2QBF. The first problem is deciding whether a quantified Boolean formula (QBF) $\Phi = \exists X \forall Y \phi$ is true, where X and Y are disjoint sets of propositional variables and ϕ is a 3DNF over $X \cup Y$ (this task is Σ_2^p-complete in general).

[1] http://www.dlvsystem.com, http://potassco.sourceforge.net/

We encode Φ as a program (with bounded predicate arities) containing a set of "guessing rules" for the X variables, and a single constraint for checking the satisfaction of the Y variables, based on predicates $c_{i,3-i}/3, i = 0, ..., 3$ for clause satisfaction. E.g., $\exists X \forall Y (x_1 \wedge \neg x_2 \wedge y_1) \vee (\neg x_2 \wedge y_1 \wedge \neg y_3)$ is encoded by the following rules

$$v(x_1, 0) \vee v(x_1, 1) \leftarrow \qquad v(x_2, 0) \vee v(x_2, 1) \leftarrow$$
$$\leftarrow v(x_1, X_1), v(x_2, X_2), c_{1,2}(X_2, X_1, Y_1), c_{2,1}(X_2, Y_3, Y_1)$$

plus all facts $c_{i,3-i}(b_1, b_2, b_3)$, $i = 0, ..., 3$, where $(b_1, b_2, b_3) \in \{0, 1\}^3$ such that the clause $z_1 \vee \cdots \vee z_i \vee \neg z_{i+1} \vee \cdots \vee \neg z_3$ is satisfied by the assignment $z_1 = b_1, z_2 = b_2, z_3 = b_3$ (thus 7 facts per i; e.g. for $i = 2$, all but $c_{2,1}(0, 0, 1)$).

2. ChainStratComp. The second problem selected is a modification of the "strategic companies" problem (Cadoli, Giovanardi, and Schaerf 1997; Leone et al. 2006), with the restrictions reported in these papers (under which the problem is Σ_2^P-complete). To the usual conditions, we add the following: given k companies $c_1, \ldots, c_k$, such that c_i controls c_{i+1} for $1 \leq i \leq k-1$, if $c_1, c_2, \ldots, c_{k-1}$ is strategic, then c_k must also be strategic. The value of the parameter k will vary and determines the size of the input instances. To the encoding reported in (Leone et al. 2006), we add the following rules for the new condition:

$$contr(C, X_i) \leftarrow controlled\_by(C, X_1, X_2, X_3), \;\; i \in [1..3]$$
$$strat(X_k) \leftarrow contr(X_2, X_1), strat(X1), \ldots$$
$$contr(X_k, X_{k-1}), strat(X_{k-1}).$$

3. Clique. The third problem is deciding whether a graph $G = \langle V, E \rangle$ contains a clique of size at least k. The encoding used is the one reported in the Introduction.

Experiment settings In our experiments, the problem to resolve was whether a given input program is consistent, i.e., has an answer set. We used an Intel Xeon 64 bit quadcore at 3.00GHz 64-bit machine with 16 GB RAM running OpenSUSE 11.0 with Linux kernel 2.6.25.20 SMP. We used DLV (with option `-n=1` for one answer set), release Oct/11/2007, GrinGo 2.0.3 and claspD 1.1 (which outputs by default one answer set). For every problem instance, we allowed a maximum of 2 hours (7200s) execution time and 2GB memory.

Test instances were randomly generated, with no attempt to produce "hard" instances. However, "trivial" instances, which have answer sets that are easily found or are trivially shown to be inconsistent, were avoided. For each test problem, one instance was generated for a combination of values of its parameters. In **2QBF**, the parameters were the size of the sets $|X|$, $|Y|$ and the number of clauses k; 40 instances were generated with $|X|$ ranging from 5 to 22, $|Y|$ from 6 to 25 and k from 5 to 32. For **ChainStratComp**, the parameters were the number of companies c, the number of products, p, and the size of the "chain" rule, k. We generated 44 instances with c ranging from 4 to 10, p from 8 to 20, and k from 5 to 15. The input graphs for **Clique** instances were randomly generated with n nodes, where n ranged from 5 to 23. At each node, 3 edges connecting the node to other (possibly not distinct) nodes were randomly generated. From these graphs, we then constructed a set of **Clique** instances, where

Figure 2: Space (left) and time (right) usage on **2QBF**

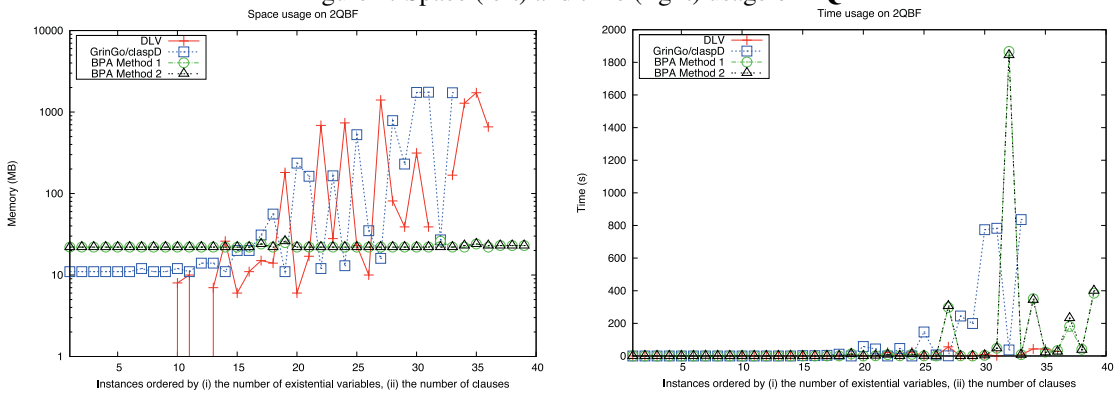


Figure 3: Space (left) and time (right) usage on **ChainStratComp**

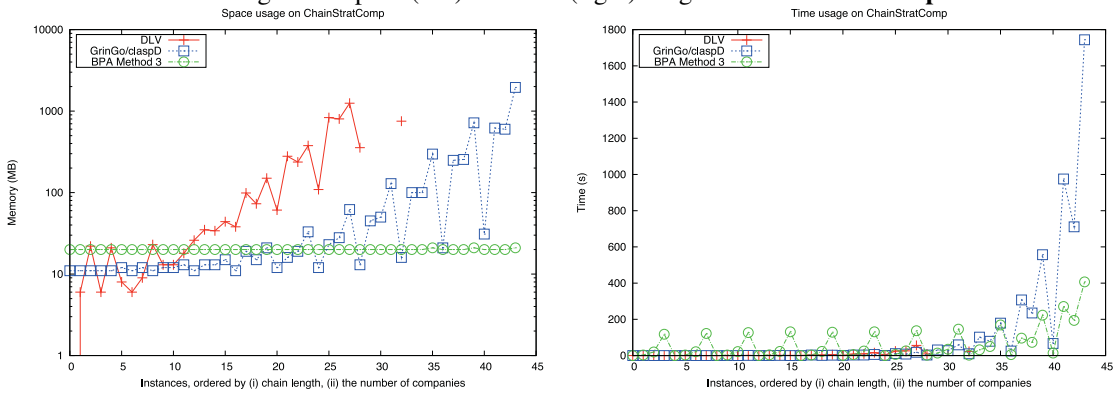
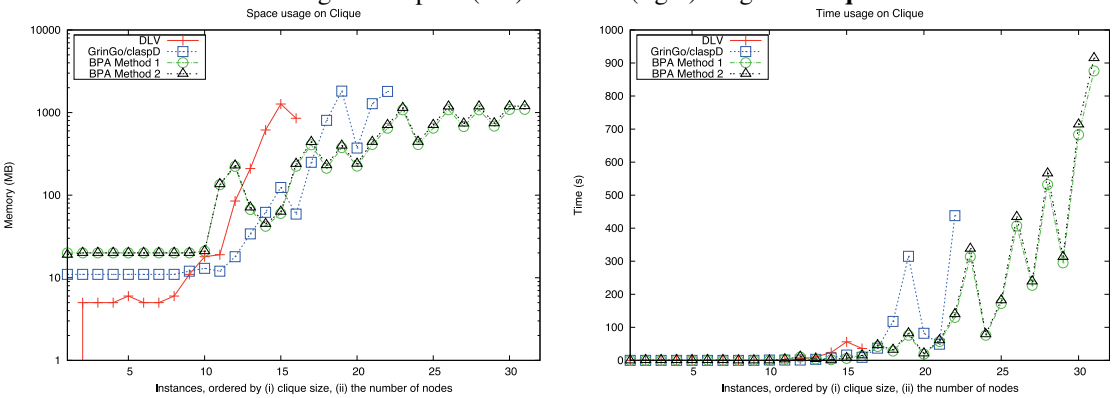


Figure 4: Space (left) and time (right) usage on **Clique**



the clique size k was chosen between $n/2$ to $n/4$. In total, we generated 32 instances of **Clique**.

Results The general observation of the test results is that BPA exhibited a polynomial growth for space consumption, while DLV and claspD showed an exponential growth. Rather unexpected, however, was that also the run time of BPA showed a clear improvement in several cases. This came as a surprise, as one would expect an increase in runtime as a trade-off for space efficiency in our methods. Summarizing, the results suggest that a more space-efficient evaluation of logic programs with bounded predicate arities is feasible.

Note that in Figures 2–4 the graphs for space usage use a logarithmic scale, while those for time usage use a linear scale. Figure 2 shows the space and time usage on the **2QBF** instances. BPA, using either Method 1 or 2, could evaluate all 40 instances with relatively small space consumption (22-26 MB) compared to DLV and GrinGo/claspD, which exceeded the 2GB limit for some instances.

Of the 44 instances of **ChainStratComp**, DLV completed only 30 instances; it exceeded the 2GB limit for most of the instances with parameter $k \geq 12$ (length of the "chain-query"). GrinGo/claspD completed all test instances (the last barely, using 1951 MB of memory and running almost 1600s). BPA completed all test instances within less memory and (in most cases) also faster.

The execution times in Figure 3 (right) show a clear advantage for BPA over DLV and GrinGo/claspD, especially for bigger instances where the effect of the exponential grounding in DLV and GrinGo/claspD impacts. The triangle pattern for BPA for the first instances is due to the instance ordering: The runtime of BPA in these areas depends mostly on the number of companies, while space and time consumption for the other systems depend mostly on the chain length.

For **Clique**, DLV completed 17 of the 32 instances and GrinGo/claspD 23, while BPA completed all 32 test instances, for both Method 1 and 2. Moreover, for growing instance size, BPA generally completed the evaluation also faster than the other systems, with Method 1 slightly ahead of Method 2; cf. Figure 4 (right). There, the missing data points for DLV and GrinGo/claspD are due to time-outs.

Even if BPA shows a much better space consumption increase than the other two systems, it still used a fairly large amount of memory in some cases (up to more than 1 GB). This is due to a memory leak of XSB in thread memory management, which is currently being worked on by the XSB developers, hence BPA has potential for lighter memory use.

Conclusion

We have presented three methods to evaluate logic programs with bounded predicate arities in polynomial space, and a framework architecture to implement the methods utilizing current ASP solvers and Prolog. Experimental results confirm the expected space usage for a prototype implementation, while ASP solvers show exponential space behavior.

Several issues remain for further work. One is to refine and enhance the methods (e.g., more sophisticated pruning of the subprograms visited, or generalizing Methods 1 and 2 to non-HCF programs). Another issue is efficient computation

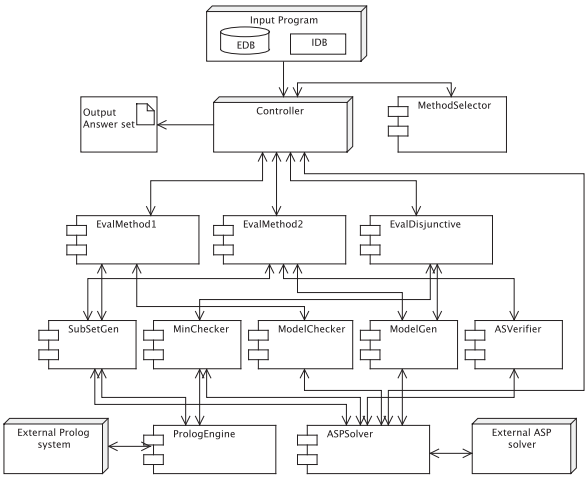


Figure 1: Framework architecture

of all answer sets (which is implicit e.g. in DLV's non-ground query answering). Finally, designing a tight integration of the methods with current ASP solvers, and in connection with this, to handle constructs like DLV's weak constraints and aggregates, or SMODELS's cardinality constraints.

References

Ben-Eliyahu, R., and Dechter, R. 1994. Propositional semantics for disjunctive logic programs. *Ann. Math. Artif. Intell.* 12:53–87.

Cadoli, M.; Giovanardi, A.; and Schaerf, M. 1997. Experimental analysis of the computational cost of evaluating quantified boolean formulae. In *Proc. AI*IA '97*, 207–218.

Drescher, C.; Gebser, M.; Grote, T.; Kaufmann, B.; König, A.; Ostrowski, M.; and Schaub, T. 2008. Conflict-driven disjunctive answer set solving. In *Proc. KR'08*, 422–432.

Eiter, T.; Faber, W.; Fink, M.; and Woltran, S. 2007. Complexity results for answer set programming with bounded predicate arities and implications. *Ann. Math. Artif. Intell.* 51(2-4):123–165.

Eiter, T.; Gottlob, G.; and Mannila, H. 1997. Disjunctive datalog. *ACM TODS* 22:364–418.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proc. ICLP'88*, 1070–1080.

Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; Perri, S.; and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM TOCL* 7(3):499–562.

Mushthofa, M. 2009. Evaluation of answer set semantics for programs with bounded predicate arities. MSc thesis, TU Vienna.

Oikarinen, E., and Janhunen, T. 2008. Achieving compositionality of the stable model semantics for smodels programs. *Theory Pract. Log. Program.* 8(5-6):717–761.

Sagonas, K.; Swift, T.; and Warren, D. S. 1994. XSB as an efficient deductive database engine. In *Proc. SIGMOD '94*, 442–453.

Schindlauer, R. 2006. *Answer-Set Programming for the Semantic Web*. Ph.D. Dissertation, TU Vienna.

Simons, P.; Niemelä, I.; and Soininen, T. 2002. Extending and Implementing the Stable Model Semantics. *AIJ* 138:181–234.