Two Applications of the ASP-Prolog System: Decomposable Programs and Multi-context Systems

Tran Cao Son, Enrico Pontelli, and Tiep Le

Department of Computer Science, New Mexico State University
{tson,epontell,tile}@cs.nmsu.edu

Abstract. This paper presents two applications of the *ASP-Prolog* system, one of the earliest modular logic programming frameworks for integrating ASP and traditional Prolog/CLP reasoning. Both applications represent significant challenges to existing ASP technologies and share some common traits—mostly related to the inadequacy of the *ground-and-solve* approach. The first application stems from several practical experiences in using state-of-the-art Answer Set Programming (ASP) solvers to tackle combinatorial problems in different domains (e.g., bioinformatics, distributed constraint problem solving). A recurrent issue is the presence of computationally tractable subproblems that turn out to be challenging, or even practically infeasible, for current ASP technologies. The second application of ASP-Prolog is its use to compute the equilibrium semantics of *Multi-Context Systems (MCS).*

1 Introduction and Motivation

Answer Set Programming (ASP) [15,13] is a declarative programming paradigm that has gained a prominent role in a variety of application domains, especially in domains with knowledge-intensive applications and combinatorial problems in high complexity classes. An important driving force behind the success of ASP is the continuous development and improvement of state-of-the-art ASP solvers, that has led to several highly competitive ASP solvers (e.g., CLASP[1]). The majority of ASP solvers employ heuristic search in computing answer sets. To facilitate the use of variables in ASP programs, ASP solvers use a two-stage approach, referred to as *ground-and-solve*, in computing answer sets of programs with variables. The program is first grounded—by replacing variables with all possible variable-free terms—and the ground program is used for the computation of solutions. ASP solvers require the program resulting from grounding to be finite. In order to accomplish this, ASP solvers impose different syntactical restrictions on programs with variables. These restrictions may disallow certain problem encodings: such encodings might represent natural ASP representations of problems, but violate some of the syntactical restrictions imposed by the ASP solvers.

There have been attempts to integrate ASP with other programming environments for different purposes (see [7] for a discussion), including early attempts to integrate ASP and Prolog (e.g., [7,4]). The ASP-Prolog system [7,17] represents one of the first

[1] http://potassco.sourceforge.net/

M. Flatt and H.-F. Guo (Eds.): PADL 2014, LNCS 8324, pp. 87–103, 2014.
© Springer International Publishing Switzerland 2014

systems proposed to provide an embedding of ASP within Prolog. ASP-Prolog is an extension of a modular Prolog system, which enables the integration of Prolog-style reasoning with ASP. The overarching goal of ASP-Prolog is to provide a platform for the integration of heterogeneous knowledge bases and an alternative computation paradigm to ASP (goal-oriented computation vs. model computation). In general, an ASP-Prolog program is a collection of modules, where each module can be declared to contain either Prolog code or ASP code. Each module provides an interface which allows the module to export predicate definitions and import definitions from other modules. In the current implementation, the root of the module hierarchy is expected to be a Prolog module, that can be interacted with using the traditional Prolog-style query-answering mechanism. ASP-Prolog has been used in several applications (e.g., [14,21]).

The objective of this paper is twofold. On one hand, we intend to identify a class of interesting problems that are challenging for existing ASP systems—and ASP-Prolog is used to illustrate a technique that can address such problems. On the other hand, the paper shows how the relatively simple features of ASP-Prolog can provide elegant and effective solutions in challenging domains.

2 Background: Logic Programming, ASP-Prolog, and MCS

Logic Programming. A logic program Π is a set of rules of the form

$$c \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n$$

where $0 \leq m \leq n$, each a_i is a literal of a propositional language[2] and $not\ a_j$, $m < j \leq n$, is called a negation-as-failure literal (or naf-literal). c can be a literal or omitted. When $n = 0$, the rule is called a *fact*. When c is omitted, the rule is a *constraint*. For a rule r, $pos(r)$ denotes the set $\{a_1, \ldots, a_m\}$ and $neg(r)$ is the set $\{a_{m+1}, \ldots, a_n\}$. A set of literals X is consistent if there is no atom a s.t. $\{a, \neg a\} \subseteq X$. A rule r is *satisfied* by X if **(i)** $neg(r) \cap X \neq \emptyset$, **(ii)** $pos(r) \setminus X \neq \emptyset$, or **(iii)** $c \in X$.

Let Π be a program. For a consistent set of literals S, the *reduct* of Π w.r.t. S, denoted by Π^S, is the program obtained from the set of all rules of Π by deleting (**i**) each rule that has a naf-literal *not* a in its body with $a \in S$, and (**ii**) all naf-literals in the bodies of the remaining rules. S is an *answer set* of Π [10] if it satisfies the following conditions: (**i**) If Π does not contain any naf-literal then S is the minimal set of literals satisfying all rules in Π; and (**ii**) If Π contains some naf-literal then S is an answer set of Π if S is the answer set of Π^S. For convenience of notation, we will use some extensions of ASP that have been proposed—such as choice atoms as defined in [20], that can occur in a rule wherever a literal can, and aggregate literals.

We will focus on programs that admit a splitting sequence[3] [11]. For a program Π, a set of literals S is a *splitting set* of Π if for every rule r of Π, if $head(r) \in S$ then $pos(r) \cup neg(r) \subseteq S$. A sequence of splitting sets $\langle S_i \rangle_{i=0}^{\infty}$ of Π is a splitting sequence of Π, if $S_i \subseteq S_j$ for $i \leq j$ and $\bigcup_{i=0}^{\infty} S_i$ is the set of literals occurring in Π.

ASP-Prolog. The ASP-Prolog system, used in this paper, has been originally described in [7,17]. It provides a modular structure and a set of predicates to enable the interaction

[2] A rule with variables is viewed as a shorthand of the set of its ground instances.

[3] For simplicity of the presentation, we consider only splitting sequences with ordinal ω.

between Prolog modules and ASP modules. Among the various components (see [17]), ASP-Prolog's interface includes:

- `use_asp(+ASPModule, +PModule, +Parameters)`: The Prolog module `PModule` is created, providing predicates to access the answer sets of the ASP program `ASPModule` with the parameters specified in `Parameters`. The new module contains the literals entailed by the skeptical semantics of `ASPModule` and has sub-modules which encode the answer sets of `ASPModule`. `PModule` provides the names of the models containing the answer sets through atoms of the form `model/1`. `PModule` and the `Parameters` arguments are optional.
- `assertnb(ASPModule, Progs)` and `retractnb(ASPModule, Progs)`: these two predicates are extended versions of the `assert` and `retract` predicates of Prolog, designed to operate on modules associated to ASP programs. Their effects are to add and remove, respectively, the clauses specified in `Progs` to the `ASPModule` and create new modules analogously to `use_asp(ASPModule)`.

Multi-context Systems (MCS). *Heterogeneous nonmonotonic multi-context systems* (MCS) have been introduced in [3]. A *logic* is a tuple $L = (\boldsymbol{KB_L}, \boldsymbol{BS_L}, \boldsymbol{ACC_L})$ where $\boldsymbol{KB_L}$ is the set of well-formed knowledge bases of L—each being a set of formulae. $\boldsymbol{BS_L}$ is the set of possible belief sets; each element of $\boldsymbol{BS_L}$ is a set of syntactic elements representing the beliefs L may adopt. Finally, $\boldsymbol{ACC_L} : \boldsymbol{KB_L} \rightarrow 2^{\boldsymbol{BS_L}}$ is a function specifying the *"semantics"* of L by assigning to each element of $\boldsymbol{KB_L}$ a set of acceptable sets of beliefs.

Using the concept of logic, we can introduce the notion of multi-context system. A *Multi-Context System (MCS)* $M = (C_1, \cdots, C_n)$ consists of contexts $C_i = (L_i, kb_i, br_i)$, $(1 \leq i \leq n)$, where $L_i = (\boldsymbol{KB_i}, \boldsymbol{BS_i}, \boldsymbol{ACC_i})$ is a logic, $kb_i \in \boldsymbol{KB_i}$ is a knowledge base, and br_i is a set of L_i-bridge rules of the form:

$$s \leftarrow (c_1 : p_1), \cdots, (c_j : p_j), not\ (c_{j+1} : p_{j+1}), \cdots, not\ (c_m : p_m)$$

where $1 \leq c_k \leq n$, p_k is an element of some belief set of L_{c_k}, $1 \leq k \leq m$, and $kb \cup \{s\} \in \boldsymbol{KB_i}$ for each $kb \in \boldsymbol{KB_i}$. Intuitively, a bridge rule r allows us to add s to a context, depending on the beliefs in the other contexts. Given a bridge rule r, we will denote by $head(r)$ the part s of r. The semantics of MCS is described by the notion of belief states. Let $M = (C_1, \cdots, C_n)$ be a MCS. A *belief state* is a sequence $S = (S_1, \cdots, S_n)$ where each S_i is an element of $\boldsymbol{BS_i}$.

Given a belief state $S = (S_1, \cdots, S_n)$ and a bridge rule r, we say that r is *applicable* in S if $p_i \in S_{c_i}$ for each $1 \leq i \leq j$ and $p_k \notin S_{c_k}$ for each $j + 1 \leq k \leq m$.

The semantic of a MCS M is defined in terms of particular belief states $(S_1, \cdots, S_n)$ that take into account the bridge rules that are applicable with respect to the given belief sets. A belief state $S = (S_1, \cdots, S_n)$ of M is an *equilibrium* if, for all $1 \leq i \leq n$, we have that $S_i \in \boldsymbol{ACC_i}(kb_i \cup \{head(r) | r \in br_i \textit{ is applicable in } S\})$.

Example 2.1. Let $M_1 = (C_1, C_2)$ where $C_i = (L_i, kb_i, br_i)$ for $i = 1, 2$, where L_i is the logic of programming under answer set semantics with $kb_1 = \{a \leftarrow not\ b; b \leftarrow not\ a\}$ and $br_1 = \{a \leftarrow (2 : d)\}$; and $kb_2 = \{c \leftarrow not\ d; d \leftarrow not\ c\}$ and $br_2 = \{d \leftarrow (1 : b)\}$. It is possible to show that M_1 has two equilibria $(\{a\}, \{c\})$ and $(\{a\}, \{d\})$.

3 Decomposable Programs

3.1 Use Cases

Use Case #1: Optimal Communication Orders between Constraint Nodes. Let us consider a *distributed constraint satisfaction problem (DisCSP)* [22]. A DisCSP is a tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A})$, where $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a finite set of variables, $\mathcal{D} = \{D_1, \ldots, D_n\}$ is a corresponding set of finite domains, $\mathcal{C}$ is a set of binary constraints $C_{i,j}$ (on the variables x_i and x_j) and $\mathcal{A} = \{A_1, \ldots, A_k\}$ is a set of agents. Each agent A_i owns a subset $\mathcal{X}_{A_i}$ of the variables of $\mathcal{X}$, s.t. $\mathcal{X}_{A_1}, \ldots, \mathcal{X}_{A_k}$ is a partition of $\mathcal{X}$. A solution to a DisCSP is a complete variable assignment satisfying all constraints.

A large number of DisCSP algorithms rely on implementing an asynchronous depth-first search (DFS). Each agent is placed in an ordering relation; the DFS is reproduced by having each agent communicate their variable instantiations towards the children agents, and in case of failure propagate backtracking to their parent agent. This assumes an ordering $\prec$ among agents—where $A_u \prec A_v$ denotes that agent A_u is the parent of A_v in the DFS tree. The ordering should ensure that if there is a constraint $C_{i,j}$ such that $x_i \in \mathcal{X}_{A_u}$ and $x_j \in \mathcal{X}_{A_v}$, then either $A_u \prec^* A_v$ or $A_v \prec^* A_u$ ($\prec^*$ is the transitive closure of $\prec$). Algorithms (e.g., [23]) have been proposed to compute such orderings.

A more complex, but realistic, scenario originates from the assumption that the communications among agents have non-uniform costs. Let us denote with $\omega(A_u, A_v)$ the communication cost between agents A_u and A_v. In this case, the goal is to determine an ordering that will minimize the maximum communication cost among agents. The communication cost between agents A_u and A_v with respect to an agent order $\prec$, denoted by $\zeta_\prec(A_u, A_v)$, such that $\zeta_\prec(A_u, A_u) = 0$ for any agent A_u, and $\zeta_\prec(A_u, A_v) = \max\{\omega(A_u, x) + \zeta_\prec(x, A_v) \mid A_u \prec x, x \prec^* A_v\}$ for any two agents $A_u \neq A_v$ such that $A_u \prec^* A_v$. The overall cost is $\zeta(\prec) = \max\{\zeta_\prec(A_u, A_v) \mid A_u \prec^* A_v\}$.

The problem admits an elegant encoding in ASP. Let us assume that the facts of the form $\texttt{edge}(X, Y)$ are used to describe the constraint graph of a DisCSP—where $\texttt{edge}(X, Y)$ states that there exists a constraint containing variables owned by X and Y. Similarly, let $\texttt{comm}(X, Y, C)$ denote that the non-negative cost of communication between agents X, Y is C. The ordering $\prec$ and the DFS tree can be generated by:[4]

$1\{root(X) : node(X)\}1.$ $\quad\quad$ $\{order(X,Y)\} \leftarrow node(X), node(Y), X \neq Y, not\ root(Y).$

The $\prec^*$ relation can be described by a simple transitive closure:

$order\_s(X,Y) \leftarrow order(X,Y).$ $\quad\quad$ $order\_s(X,Z) \leftarrow order(X,Y), order\_s(Y,Z).$

The following constraints guarantee the DFS conditions:

$\leftarrow order(X,Y), order(Y,X).$ $\quad\quad$ $\leftarrow node(Y), 2\{order(X,Y) : node(X)\}.$

$\leftarrow node(X), not\ root(X), not\ has\_ancestor(X).$

$\leftarrow edge(X,Y), \{order\_s(X,Y), order\_s(Y,X)\}0.$

We can associate costs to agents based on their distance from the root of the DFS:

$cost\_node(X, 0) \leftarrow root(X).$

$cost\_node(Y, C1 + C2) \leftarrow not\ root(Y), order(X,Y), comm(X,Y,C1), cost\_node(X, C2).$

$leaf\_cost(X, C) \leftarrow cost\_node(X, C), leaf(X).$

[4] We omit the definition of trivial predicates like `leaf`.

and the cost of the resulting DFS tree is

$$tree\_cost(C1) \leftarrow C1 = \#\max[leaf\_cost(\_, C) = C], C1 > 0.$$

Note that the definition of *tree_cost* makes use of an aggregate (`#max`). Our objective is to determine DFSs with certain properties, e.g., with minimal cost. Observe that this program has a splitting sequence S_0, S_1, S_2 where:

(i) S_0 consists of literals of the form $node(X)$, $edge(X,Y)$, $leaf(X)$, $root(X)$, $order(X,Y)$, and $order\_s(X,Y)$;
(ii) S_1 consists of S_0 and the literals relating to the cost ($cost\_node(X,Y)$ and $leaf\_cost(X,Y)$); and
(iii) S_3 consists of S_2 and the other literals.

Intuitively, this splitting sequence represents the steps involved in the process of solving the problem: *(i)* create an ordering among the nodes; *(ii)* compute the cost of each leaf of the specified order; *(iii)* compute the cost of the tree as the maximal cost of the leaves.

The difficulty posed to ASP solvers by the above program lies in the rules defining the cost of the nodes. Without bounding the cost `C2` of the rule, the grounding process does not terminate in any reasonable amount of time. Limiting the cost `C2` to be the sum of all (positive) costs allows CLASP to solve instances that have a small overall communication cost between nodes (e.g., when the the bound to `C2` is smaller than $1,000$). The grounder needs to ground all combinations of the second rule in this group and this number increases with the bound. We observe that the system ASP{f} [2] could be useful in this situation. An alternative approach to using functions is to *inform the grounder* about the steps in computing the answer sets of the program via an extra parameter (i.e., effectively exposing the stratification to the grounder):

$$cost(X, 0, 1) \leftarrow root(X).$$
$$cost(X, C1 + C2, T + 1) \leftarrow level(T), not\ root(X), order(Y, X), comm(Y, X, C1), cost(Y, C2, T).$$
$$leaf\_cost(Y, C) \leftarrow leaf(Y), cost(Y, C, T + 1), level(T).$$

where `level` is a predicate defining the level in a tree, which can be at most the number of nodes in the graph. With this change, CLASP is able to identify that `C2` can only take a small set of possible values and has no problem with the value of the weights. The potential cyclic dependency between `cost(X,C1+C2)` and `cost(Y,C2)` is now a single way dependency (i.e., first depends on the second). Although the method of breaking dependencies works in this problem, it does not work in the next example.

Use Case #2: Approximated Supertree Computation. We consider *phylogenies* [12] as trees where each internal node has at least two children. We will assume the traditional terminology for trees. For a tree T, let $L(T)$ denote the set of leaves of T. An *internal edge* is an edge connecting two internal nodes, one of which can be the root. A *cluster* is the set of all the leaves that are descendants of the same internal node. Let us denote with *MRCA*(S) the most recent common ancestor of the set of leaves S.

For two sets of leaves $A, B \subseteq L(T)$, $A <_T B$ if *MRCA*(A) is a descendant of *MRCA*(B). A tree T' is obtained from T by *contraction* if T' can be obtained from T by contracting some internal edges. Let $A \subseteq L(T)$. The subtree of T with the leaf set A is the subtree of T whose root has A as its cluster; we refer to it as the *subtree of*

T *induced by* A, and denote it with $T|A$. A tree T *displays* a tree t if t is an induced subtree of T, or can be obtained from an induced subtree by contraction.

Let $\mathcal{T}$ be a collection of trees and $S = \bigcup_{T \in \mathcal{T}} L(T)$. A *supertree method* takes $\mathcal{T}$ as input and returns a tree T with the leaf set S such that T displays each element of $\mathcal{T}$ [18]. Several popular algorithms to compute supertrees have been proposed; in this section, we consider a rough approximation of the method in [19]. The approximation has been developed to quickly generate putative supertrees as part of the CDAOStore project [16]. For a tree T and a set of leaves S, $pruned(T, S)$ denotes the tree obtained from T by: **(1)** Deleting all the subtrees of internal nodes whose set of leaves is a subset of S, and the edges coming into these internal nodes; **(2)** Deleting all the remaining leaves appearing in S and the edges leading to such leaves; and **(3)** Simplifying the remaining tree by removing all internal nodes which have only one child. A *weighted tree* T is a tree with an associated weight w. Given $\mathcal{T}$, a weighted graph $S_{\mathcal{T}}$ is defined as follows: **(1)** The nodes of $S_{\mathcal{T}}$ are the leaves of $\mathcal{T}$; **(2)** Nodes a and b are connected if a and b are in a proper cluster in one of the trees in $\mathcal{T}$ (i.e., if there is a tree in $\mathcal{T}$ where $MRCA(a, b)$ is not the root of the tree); **(3)** The weight of an edge (a, b) in the graph $S_{\mathcal{T}}$ is the total weight of all trees in which a and b are in a proper cluster.

The APPROXSUPERTREE algorithm is described in Algorithm 1. This program runs in polynomial time in the size of the trees. The APPROXSUPERTREE algorithm can be

Algorithm 1. APPROXSUPERTREE($\mathcal{T}$)

Require: a set of k trees $\mathcal{T}$, with leaves set $S = \bigcup_{T \in \mathcal{T}} L(T) = \{x_1, \ldots, x_n\}$.
1: **if** n = 1 or n = 2 **then**
2: **return** a single node labeled by x_1 or x_1 and x_2
3: **end if**
4: construct $S_{\mathcal{T}}$
5: **if** $S_{\mathcal{T}}$ is connected **then**
6: Let E^{cut} be the set edges of minimal weight of $S_{\mathcal{T}}$
7: $S_{\mathcal{T}}/E^{cut}$ is obtained from $S_{\mathcal{T}}$ by deleting all edges in E^{cut}
8: Replace $S_{\mathcal{T}}$ with $S_{\mathcal{T}}/E^{cut}$
9: **end if**
10: Let $S_1, \ldots, S_k$ be the components of $S_{\mathcal{T}}$
11: **for** each component S_i **do**
12: $T_i =$APPROXSUPERTREE($\mathcal{T}|S_i$), where $\mathcal{T}|S_i = \{pruned(T, L(T) \setminus S_i) \mid T \in \mathcal{T}\}$
13: Construct a new tree T' by connecting the roots of the trees T_i to a new root r
14: **end for**
15: **return** T

implemented by an ASP program with the following basic components that implement one iteration of the algorithm. To fully implement this algorithm, the predicates need to be extended with an extra parameter denoting the iteration step.

- *Encoding trees:* a tree is described by a set of atoms of the form $edge(t, n_1, n_2)$ and a fact $tree(tree\_name, weight)$. Rules for defining node, root, leaf, ancestor (*anc*), etc. can be easily defined based on these predicates and are omitted to save space.
- *Code for computing the pruned tree:* this code computes the tree $pruned(T, L(T) \setminus S_i)$ (Line 12, Algorithm 1). We assume that the tree T and the pruned set of

leaves S are given. Elements of S are specified by $member(X, S)$. First, we define $some\_descendants\_out(T, S, N)$ which is true whenever N—a non-leaf in the tree T—has a descendant that does not belong to the pruned set S.

$$some\_descendants\_out(T, S, N) \leftarrow pruned\_set(S), node(T, N), leaf(T, N_1), \\ not\ member(N_1, S), ancestor(T, N, N_1).$$

We then identify the nodes that should be deleted. These are the nodes whose leaf-descendants belong to the pruned set.

$$delete\_node(T, S, N) \leftarrow pruned\_set(S), node(T, N), not\ leaf(T, N), \\ not\ some\_descendants\_out(T, S, N).$$

The above predicates are used to define the predicate $simplify\_node(T, S, N)$, which says that a non-leaf node that has not been deleted should be simplified if it has only one child that is not deleted.

$$simplify\_node(T, S, N) \leftarrow pruned\_set(S), node(T, N), not\ leaf(T, N), \\ not\ delete\_node(T, S, N), \\ NC = \#count\{edge(T, N, N_1) : not\ delete\_node(T, S, N_1) \\ : not\ member(N_1, S)\}, NC < 2$$

A node remains after pruning if it is not deleted, not a member of the pruned set, and not simplified. This is defined by the following predicate.

$$is\_new\_node(T, S, N) \leftarrow pruned\_set(S), node(T, N), not\ member(N, S), \\ not\ delete\_node(T, S, N), not\ simplify\_node(T, S, N).$$

This allows us to define the new tree $pruned(T, S)$, that is the result of pruning S from the tree by identifying the edges of the tree.

$$ptree\_edge(pruned(T, S), N_1, N_2) \leftarrow pruned\_set(S), tree(T, W), ancestor(T, N_1, N_2), \\ is\_new\_node(T, S, N_1), is\_new\_node(T, S, N_2), NA < 1, \\ NA = \#count\{ancestor(T, NM, N_2) : ancestor(T, N_1, NM) \\ : not\ simplify\_node(T, S, NM)\}.$$

This rule says that there is an edge between two nodes N_1 and N_2 in the tree, after pruning, if N_1 is an ancestor of N_2 and all the ancestors of N_2 which are descendants of N_1 have been simplified.

- *Code for computing the connected graph of leaves of a set of trees*: this code creates the weighted graph $\mathcal{S}_\mathcal{T}$ (Line 4, Algorithm 1). We first identify the cluster of a tree. Two leaves A and B of the tree T with the weight W are in a proper cluster if they share the same ancestor which is not the root.

$$in\_cluster(T, A, B, W) \leftarrow tree(T, W), leaf(T, A), leaf(T, B), node(T, N), \\ not\ root(T, N), ancestor(T, N, A), ancestor(T, N, B).$$

Next we define the edge of the graph. An edge of the graph is an edge between two nodes in the same cluster. The weight of the edge is the sum of all the weights of the corresponding trees in which the nodes appear in the same cluster.

$$graph(A, B, WG) \leftarrow leaf(T, A), leaf(T, B), \\ 1\{in\_cluster(T_1, A, B, W) : tree(T_1, \_)\}, WG = \#sum[in\_cluster(\_, A, B, W) = W].$$

- *Code for constructing the supertree after one iteration:* this code accomplishes the task of computing T_i (Line 12, Algorithm 1). This starts with the construction of the reduced graph by eliminating edges with minimal weight. To achieve this, we compute the minimal weight (WM).

$$min\_edge(WM) \leftarrow WM = \#min[graph(\_, \_, W_1) = W_1].$$

The reduced graph will contain edges whose weight is greater than the minimal weight. We also introduce the predicate $not\_reduced(A)$, to indicate that A is the vertex of at least one edge that is not eliminated.

$$3\{not\_reduced(A), not\_reduced(B), reduced\_graph(A,B,W)\}3 \leftarrow graph(A,B,W), min\_edge(WMin), WMin < W.$$

If all edges going out from a vertex are eliminated then the vertex is itself a component of the reduced graph. We have the following rule to characterize this.

$$reduced\_graph(A,A,0) \leftarrow graph(A,B,W), min\_edge(WMin), WMin == W, not\ not\_reduced(A).$$

The next step in computing the supertree is to identify the components of the reduced graph and select a representative for each component. The rule

$$\{representative(A)\} \leftarrow leaf(T,A).$$

defines that only leaves could be selected to be representatives. Since each component has one and only one representative, we add the following rules:

$$connected(A,B) \leftarrow 1\{reduced\_graph(A,B,\_), reduced\_graph(B,A,\_)\}.$$
$$connected(A,B) \leftarrow reduced\_graph(A,C,\_), connected(C,B).$$
$$is\_connected(A) \leftarrow representative(B), leaf(T,A), connected(A,B).$$
$$is\_connected(A) \leftarrow representative(B), leaf(T,A), connected(B,A).$$
$$\leftarrow leaf(T,A), not\ is\_connected(A).$$
$$\leftarrow representative(A), representative(B), A \neq B, connected(B,A).$$

The first four rules define the connectivity relationship, based on the edges of the reduced graph. The next rules guarantee that exactly one representative is selected for each component. Having computed the components and their representatives, the supertree can be derived using the following rules. To identify the number of nodes in a component, we define the degree of a node using the rule:

$$degree(A,D) \leftarrow representative(A), D = \#count\{connected(A,\_)\}.$$

If the component is a singleton then it will be connected to the root, which is represented by the name of the pruned set.

$$edge\_s(S,A) \leftarrow representative(A), degree(A,1), pruned\_set(S).$$

If the component has exactly two elements, then a new root will be created and connected to the root. The new root is connected to the two leaves. In the next rule, $@newName(S,A)$ creates a new constant that is a direct descendant of the pruned set and is the parent of the two leaves.

$$3\{edge\_s(S,N), edge\_s(N,A), edge\_s(N,B)\} \leftarrow representative(A), degree(A,2), connected(A,B), A \neq B, pruned\_set(S), N := @newName(S,A).$$

If the component has more than two elements then a new root is created and the algorithm computes the pruned set for the next iteration.

$$2\{edge\_s(root,N), generated\_pruned\_set(N,A)\} \leftarrow representative(A), degree(A,D), D > 2, pruned\_set(S), N := @newName(S,A).$$

Here, $generated\_pruned\_set(N,A)$ records a new pruned set, named N, related to the representative A. The pruned set and its members are defined in the next rules.

$$new\_pruned\_set(N) \leftarrow generated\_pruned\_set(N,A).$$
$$new\_member(B,N) \leftarrow generated\_pruned\_set(N,A), ptree\_leaf(T,B), not\ connected(A,B).$$

Observe that $ptree\_leaf$ is defined in a similar fashion as $leaf$.

Let us point out the following aspects that prevent an effective use of ASP for this algorithm. The algorithm needs to be repeated until all components contain either one or two leaves. In each iteration, the computation needs to identify the set of leaves that will be pruned for the computation of the pruned trees. Given the set of leaves that will be pruned, the pruned trees are uniquely identified. On the other hand, if the set of leaves is unknown, the ASP solver will need to guess, i.e., it will have to ground *all possible combinations* of the leaves of the trees. Thus, the proposed encoding can only deal with problems with very few leaves (e.g., less than 10). A simple example commonly encountered in the literature (12 leaves) cannot be solved. We observe that the full ASP implementation of this algorithm also possesses a splitting sequence corresponding to the iteration steps that the algorithm must go through in order to compute the supertree.

Use Case #3: Union of All MinCut Sets of a Weighted Graph. A MinCut of a weighted graph is a set of edges of minimal weight that disconnects the graph. The problem we consider is to compute the union of all MinCuts of a graph (useful, e.g., for supertree computations). Unlike the previous problems, this problem has a polynomial time algorithm but does not seem to have a straightforward ASP encoding.

If we only need to compute one MinCut, the ASP encoding is simple: generate a cut and minimize its weight. The problem is no longer trivial when we need to generate the union of all MinCuts. One can try to index the possible MinCuts and use multiple minimization statements. Since the minimal weight is unique, one would have to add a constraint that the weight of the cuts is unique. This encoding faces several problems, e.g., it requires the number of MinCuts and the solver will try to find an answer set where all weights are equal—which is not necessarily the minimal weight.

An alternative approach relies on the observation that, given a graph $G = (V, E)$, an edge $e \in E$ is in at least one MinCut of G iff $c(G) = c((V, E \backslash \{e\})) + w(e)$, where $c(G)$ is the cost of the MinCut of G and $w(e)$ is the weight of edge e. This can be captured by distinct sets of rules that compute one MinCut for G and for each $(V, E \setminus \{e\})$, plus a final rule that checks which edges have the above property. In this case, grounding is not an issue, the repeated minimizations required to compute MinCuts lead to a very large computation time (no results after 2 hours), while the individual sets of rules can be executed in less than one second. Also in these cases, an interleaved grounding and solving would help, by allowing the accumulation of MinCuts from iteration to iteration, and combining the weights of MinCuts at the end to determine relevant edges.

3.2 ASP-Prolog for Decomposable Programs

Interleaving Grounding and Computation. The above three examples highlight a real limitation of ASP solvers that employ the traditional ground-and-solve approach. All three problems share a property that each program possesses a splitting sequence corresponding to the steps that can be used in computing the answer of the problem. This is characterized by the splitting sequence theorem in [11]. The theorem shows that, for each answer set A of a program Π that has a splitting sequence $\langle S_i \rangle_{i \geq 0}$, there exists a decomposition of A in a sequence of sets $\langle A_i \rangle_{i \geq 0}$, such that A_i's can be computed step-by-step in the following fashion: **(1)** Compute an answer set A_0 of the bottom program $b_{S_0}(\Pi)$ that consists of all rules in Π whose atoms belong to S_0; **(2)** For each

$i \geq 0$, compute an answer set A_{i+1} of the program $e_{S_i}(b_{S_{i+1}}(\Pi) \setminus b_{S_i}(\Pi), \bigcup_{j \leq i} A_j)$. In this definition, $e_S(\Pi, X)$ is a program containing rules determined as follows: *(a)* remove all rules $r \in \Pi$ such that either there is a positive atom in the body belonging to S but not X or a negative atom belonging to S and X; and *(b)* removing all occurrence of a or $not\ a$ for $a \in S$ from the remaining rules. We refer to programs that admit a splitting sequence as *decomposable programs*.

Decomposable programs can be seen as a sequence of *lp-functions* [9] $\langle \Pi_i \rangle_{i \geq 0}$ where each lp-function Π_i accepts a set of input predicates In_i and defines a set of output predicates Out_i, such that $In_i \subseteq \bigcup_{j<i} Out_j$ for every i. Under this view, decomposable programs are well-suited to encode iterative algorithms or dynamic programming algorithms, that frequently occur in a variety of application domains.

Observe that every stratified program, whose answer sets can be computed in polynomial time in the size of the program, is decomposable—while the converse does not necessarily hold. Thus, computing answer sets of a decomposable program is not necessarily a simple task. However, the computation of an answer set of a decomposable program could potentially be done more efficiently, if the splitting theorem was applied in the process. This is because the size of the program Π_i depends not only on the original program Π, but also on the answer sets computed up to that point (i.e., $\bigcup_{j<i} A_j$). This requires the interleaving of grounding and solving. By interleaving grounding and solving, some problems that cannot be solved with current ASP solvers may become efficiently solvable—as the examples illustrated earlier in this paper.

This type of computation is quite natural to encode in the context of ASP-Prolog. Assume that the program Π has been decomposed into a list L of components. The following Prolog predicate can be used in ASP-Prolog to compute answer sets of Π.

$$solve([], Out, Out).$$
$$solve([H|T], In, A) \leftarrow use\_asp(H, H, In), H : model(M), collect\_facts(M, F), solve(T, F, A).$$

To compute answer sets of the program Π with the list of components L, the goal `solve(L, [], A)` should be issued. The predicate `collect_facts(M,F)` collects in a list `F` all elements of the answer set named `M`. Observe that the above implementation requires a prior decomposition of the program. The implementation could be improved by introducing a module that analyzes the program and automatically identifies the splitting sequence—a topic of future work.

Computing Iterative Algorithms. We will continue with a general methodology for the implementation of iterative algorithms such as the supertree computation algorithm detailed earlier. Observe that this type of algorithm can be characterized by a sequence of values $F(0), F(1), \ldots, F(n), \ldots$. The computation stops when a boolean condition, denoted by H, is satisfied. A generic procedure for computing iterative algorithms can be roughly described as follows: **(1)** initialize settings and initialize counter i to 0; **(2)** while the halting condition H is not satisfied, compute $F^i(\boldsymbol{v})$ using input $F^{i-1}(\boldsymbol{v})$ and increment i; and **(3)** return $F^i(\boldsymbol{v})$.

Assume that the initialization (step **(1)** and the body of the loop in **(2)** can be implemented by the ASP programs R and Q, such that Q has a splitting set S. This assumption is, for example, met in the supertree computation problem—thus, we can view algorithm 1 as a typical iterative algorithm and its implementation

satisfies this condition—where the splitting set S of Q contains all but the atoms of the form $new\_pruned\_set(n)$, $new\_member(b, n)$, and $generated\_pruned\_set(n, a)$ in Q. Under the above assumptions, we can use the following steps to implement the iterative algorithm in ASP: **(a)** add t as the last parameter for every predicate in S; **(b)** add $t + 1$ as the last parameter for every predicate not in S; and **(c)** add the declaration that t is a constant in Q. The following pseudo code realizes the algorithm in ASP-Prolog:

$$solve(Q, R, A) \leftarrow solve(Q, R, A, 0).$$
$$solve(Q, R, A, 0) \leftarrow use\_asp(R, R, []), R : model(M), create\_file(M, File),$$
$$(satisfied(M, H) \rightarrow A = File; solve(Q, A, File, 1)).$$
$$solve(Q, A, Prev, T) \leftarrow T > 0, create\_string(Paras, T, Prev),$$
$$use\_asp(Q, Q, [Paras]), Q : model(M), create\_file(M, File),$$
$$(satisfied(M, H) \rightarrow A = File; solve(Q, A, File, T + 1)).$$

In the above code, `create_string(Paras,T,Prev)` creates a string of the form '-c t=@T @M' where @T (@M) is replaced by the value of T ($Prev$) respectively; and the predicate `satisfied(M,H)` indicates whether the current answer set (described by module named `M`) satisfies the halting condition H on Line 3. This code is problem specific and needs to be instantiated by the programmer. A simple way to achieve this can be realized by adding a rule of the following form

$$incomplete \leftarrow not\ H.$$

to the program Q. For instance, for the program computing the supertree, the rule

$$incomplete \leftarrow new\_pruned\_set(N).$$

can be used. In this case, the test `satisfied(M,H)` is equivalent to checking the membership of $incomplete$ in `M`. We have applied this method in solving the problem of computing the approximated supertree. We should note that, with this method, we were able to compute the solutions for all three problems described in the three use cases. In particular, we can solve the largest problem for computing the supertree that was discussed in the literature (two trees, with 41 leaves and 31 leaves, respectively).

Before we conclude this section, we would like to point out that there are easy ways to facilitate an interleaving between grounding and solving using current ASP solvers (e.g., using a scripting language). However, we believe that an off-the-shelf ASP solver with this feature would have a much larger impact, as it would open ASP to other types of applications that have not been considered so far. Furthermore, we observe that this feature could be implemented in a similar fashion as the ICLINGO system (as a matter of fact, it could be a minor modification of ICLINGO).

4 Computing Equilibria

In this section, we will present another challenging application of ASP-Prolog. We describe a system, called ASP-Prolog^{MCS}, for computing the equilibrium semantics of multi-context systems (MCS). Observe that the previous applications are concerned with one program that can be decomposed into a sequence of programs, whose answer sets can be computed sequentially. The second application is concerned with a set of inter-connected programs whose semantics (an equilibrium) is a sequence of models. In many cases, these models might not be computed sequentially as in the first application. Before we detail the implementation of ASP-Prolog^{MCS} let us discuss the algorithms that can be used in computing the equilibrium semantics of a MCS.

Let $M = (C_1, \ldots, C_n)$ be a MCS, where the logic for each context is logic programming under answer set semantics [10]. For each context $C_i = (L_i, kb_i, br_i)$ and for each bridge rule r in br_i, we introduce a new set of "tagged" atoms in the language of L_i, of the form $t(c_k, p_k)$, for $k = 1, \ldots, m$ and $c_k \neq i$. We define the program $R(r)$ that consists of the following rules (for $k = 1, \ldots, m$ and $c_k \neq i$):

$$0\ \{t(c_k, p_k)\}\ 1 \leftarrow$$
$$s \leftarrow t(c_1, p_1), \ldots, t(c_j, p_j),\ \textit{not}\ t(c_{j+1}, p_{j+1}), \ldots,\ \textit{not}\ t(c_m, p_m).$$

where (c_i, p_i) is replaced by $t(c_i, p_i)$. We denote the second rule above by $t(r)$. Let $P_i = kb_i \cup \bigcup_{r \in br_i} R(r)$ and $MP = (P_1, \ldots, P_n)$. The set of literals of the form $t(k, p)$ occurring in P_i is denoted by T_i. For the MCS M_1 in Example 2.1, we have that $T_1 = \{t(2, d)\}$ and $T_2 = \{t(1, b)\}$.

For a belief state $S = (S_1, \ldots, S_n)$ of M, we define $t(i, S) = \{t(j, p) | t(j, p) \in T_i \textit{ and } p \in S_j\}$. We can show that a belief state $S = (S_1, \ldots, S_n)$ is an equilibrium of M if $S_i \cup t(i, S)$ is an answer set of P_i. Continuing with the MCS M_1 in Example 2.1, $S = (\{a\}, \{d\})$ is an equilibrium of M_1 because we have that $t(1, S) = \{t(2, d)\}$ and $t(2, S) = \emptyset$; in this case, $\{a\} \cup t(1, S)$ is an answer set of P_1, and $\{d\} \cup t(2, S) = \{d\}$ is an answer set of P_2. On the other hand, $S = (\{b\}, \{d\})$ is not an equilibrium of M_1. We can observe that $t(1, S) = \{t(2, d)\}$ and $t(2, S) = \{t(1, b)\}$; thus, $\{d\} \cup t(2, S) = \{d, t(1, b)\}$ is an answer set of P_2, but $\{b\} \cup t(1, S) = \{b, t(2, d)\}$ is not an answer set of P_1.

Two answer sets Z_i and Z_j of P_i and P_j are *compatible* if: $t(i, p) \in Z_j$ iff $p \in Z_i$ and $t(i, p) \in T_j$, and $t(j, p) \in Z_i$ iff $p \in Z_j$ and $t(j, p) \in T_i$. Again, consider the MCS M_1 in Example 2.1, $\{a\}$ is compatible with $\{c\}$; $\{a, t(2, d)\}$ is compatible with $\{d\}$; however, $\{a\}$ is not compatible with $\{d\}$.

We can show that for a sequence of answer sets $Z = (Z_1, \ldots, Z_n)$ of $(P_1, \ldots, P_n)$, $Z' = (Z_1 \setminus T_1, \ldots, Z_n \setminus T_n)$ is an equilibrium of M if Z_i is compatible with Z_j for every pair of $i \neq j$. This enables a naive computation of the equilibrium in a generate-and-test fashion: *(i)* Generate a belief state $Z = (Z_1, \ldots, Z_n)$ of $(P_1, \ldots, P_n)$; *(ii)* Check for compatibility of Z. This is the first algorithm that we implemented in ASP-Prolog^{MCS}.

The naive algorithm, however, requires an excessive amount of memory when dealing with programs that have a large number of answer sets. We can exploit the compatibility between answer sets of the programs P_i's in the construction of an equilibrium $S = (S_1, \ldots, S_n)$ of M in an incremental fashion.[5] Given a MCS $M = (C_1, \ldots, C_n)$, the algorithm needs to first compute $MP = (P_1, \ldots, P_n)$ and then compute a sequence of compatible answer sets $Z =$ `compatible`(MP). If Z is not a failure, then the result will be $S = (Z_1 \setminus T_1, \ldots, Z_n \setminus T_n)$. The function that computes a sequence of compatible answer sets (`compatible`) is given in Algorithm 2. Observe that the algorithm has two non-deterministic choices.

Both the naive and the incremental algorithms can compute MCS with arbitrary topologies, and they can be easily implemented in ASP-Prolog. Before discussing this, we consider some enhancements that take into consideration the topology of MCS. We define the *dependency graph* $G_M = (V_M, E_M)$ of a MCS $M = (C_1, \ldots, C_n)$ as follows:

- The set of vertices is $V_M = \{1, \ldots, n\}$;
- $(i, j) \in E_M$ if $(i : p)$ appears in the body of some bridge rule in br_j and $i \neq j$.

[5] This is possible since M is reducible [3].

Algorithm 2. `compatible`(MP)

1: **Input**: $MP = (P_1, \ldots, P_n)$
2: Nondeterministically select Z in `compatible`$(P_1, \ldots, P_{n-1})$
3: Assume that $Z = (Z_1, \ldots, Z_{n-1})$
4: **if** $Z \neq$ **fail then**
5: Let $Q_i^1 = \{\leftarrow not\ p \mid t(n,p) \in Z_i\}$ for $i < n$
6: Let $Q_i^2 = \{t(i,p) \leftarrow \mid t(i,p) \in T_n$ and $p \in Z_i\}$ for $i < n$
7: Let $Q_i^3 = \{\leftarrow t(i,p) \mid t(i,p) \in T_n$ and $p \notin Z_i\}$ for $i < n$
8: $P_n = P_n \cup \bigcup_{i=1}^{n-1}(Q_i^1 \cup Q_i^2 \cup Q_i^3)$
9: Select an answer set Z_n of P_n, Z_n compatible with Z_j for $j \leq n-1$
10: **return** $Z = (Z_1, \ldots, Z_{n-1}, Z_n)$
11: **end if**
12: **return fail**

Intuitively, an edge (i,j) in E_M indicates that if $S = (S_1, \ldots, S_n)$ is an equilibrium of M then the applicability of a bridge rule in br_j depends on the belief set S_i. It is well-known that the graph G_M specifies a topology that can be used in computing equilibria of M. For instance, if (i,j) is an edge in G_M and G_M does not contain the edge (j,i), it is sensible to compute the i^{th} belief set before computing the j^{th} belief set. This has been utilized in the systems DMCS and DMCSOPT [6,1].[6]

Let us define an ordering $\prec_M$ between the contexts, where $i \prec_M j$ if there exists a path from i to j in G_M. We consider the following cases:

- $\prec_M$ is a partial order: it can be extended to a total order $\prec_M^*$ over the set $\{1, \ldots, n\}$.
- $\prec_M$ contains a cycle: let $SCC_1, \ldots, SCC_t$ be a set of strongly connected components (SCC) of G_M, $SCC_i = (V_i, E_i)$, such that $V_M = \bigcup_{i=1}^{t} V_i$ and $E_M = \bigcup_{i=1}^{t} E_i$. Furthermore, the following induced order is a partial order over the SCCs: $SCC_i \prec_M SCC_j$ iff there exists some $s_i \in SCC_i$ and $s_j \in SCC_j$ such that $s_i \prec_M s_j$.

The order $\prec_M$ can be used for computing the equilibria of M as follows:

- Add the computation of the dependency graph G_M, the SCCs of G_M, and the ordering $\prec_M$ before the computation of *MP*.
- Sort $P_1, \ldots, P_n$ using $\prec_M$ on the SCCs of G_M and provide Algorithm 2 with the ordering $\prec_M$ (for programs in the same SCC, an arbitrary order is used).
- Modify Algorithm 2 to eliminate the compatibility checking between Z_n and Z_i if $n \prec_M i$ does not hold (Line 9).

The next section discusses the implementation.

4.1 ASP-Prolog^{MCS}

The current implementation of ASP-Prolog^{MCS} computes equilibria for MCS of the form $M = (C_1, \ldots, C_n)$ where the logic underlying each context is logic programming

[6] In these systems, the dependency graph is defined in reverse order and used somewhat differently from our proposal.

under answer set semantics [10]. Extending ASP-Prolog^{MCS} to allow different semantics can easily be introduced since ASP-Prolog can support different types of semantics for different modules. We assume that each $C_i = (L_i, kb_i, br_i)$ is stored in a file named p_i containing $p_i = kb_i \cup br_i$. The main predicates of ASP-Prolog^{MCS} are:

- `load_MCS(Input)`: this predicate prepares the computation of the equilibrium of the MCS whose contexts are specified by the list `Input`. Its execution will:
  - Create a Prolog module named p_i, for each $p_i \in$ `Input`, and compute $P_i = kb_i \cup \bigcup_{r \in br_i} R(r)$ (as defined earlier);
  - Create a dependency graph between contexts. This is achieved by defining a predicate $dependency/4$ where, for each literal $(c_k : p_k)$ in a bridge rule r of the context c_i such that $c_k \neq c_i$, we assert the atom $dependency(c_i, s, c_k, p_k)$. Intuitively, the atom $dependency(i, a, j, p)$ states that there is a dependence of atom a in the context i to atom p in the context j.
- `compute_equilibrium(Input, Answer)`: two versions of this predicate have been implemented. The first one implements the naive generate-and-test algorithm and the second one implements the modified algorithm discussed above. The implementation of this predicate makes use of the infrastructure provided by ASP-Prolog. Execution of this predicate will: load the files in `Input`, which is a list of files representing the MCS, compute the answer sets of each program in `Input` (via the predicate `use_asp/3`), and call the predicate that implements the naive algorithm or the Algorithm 2 to obtain the equilibrium.
- `generate_and_test(Input, Answer)`: this predicate implements the generate and test algorithm and returns the answer.
- `compatible(Input, Answer)`: this implements the algorithm `compatible` and returns the answer.

4.2 Experiments

We experimented ASP-Prolog^{MCS} with the set of benchmarks downloaded from the DMCS system website.[7] The benchmarks include five domains (`Diamond, Ring, Zig-zag, House` and `Binary tree`). The name of each domain characterizes the topology of the MCS, for example, in an instance of the `Diamond` domain, the contexts are combined by multiple diamonds in a row.

Both algorithms were used in testing this set of problems. The experiments were successful, showing a competitive performance. The only limitation encountered was in problems where selected contexts have a large number of answer sets—e.g., several thousands—in which case the answer sets occasionally saturated the streams created by the underlying Prolog system—SICStus Prolog, used in the current implementation, imposes limitations on the number of concurrent open streams (around 200 streams). Only the generate-and-test algorithm can successfully solve all problems, due to the limitations on the numbers of opened streams; the second algorithm cannot be used for MCS with more than 200 contexts. However, whenever possible both algorithms performed well. Overall, ASP-Prolog^{MCS} performs well in the search for an

[7] `www.kr.tuwien.ac.at/research/systems/dmcs/experiments.html`

equilibrium after the contexts have been loaded. The complete evaluation can be found at `www.cs.nmsu.edu/~tile/aspmcs/experiment.html`.

4.3 Application of ASP-Prolog^{MCS}

Although ASP-Prolog^{MCS} was developed for MCSs, an interesting by-product of the system is that it can be used to compute answer sets of decomposable logic programs.

Let Π be a decomposable logic program. For simplicity, let us assume that Π has a splitting set S. Let us consider a partition (Π_1, Π_2) of Π, such that Π_1 is the bottom of Π with respect to S, i.e., $\Pi_1 = b_S(\Pi)$ and $\Pi_2 = \Pi \setminus \Pi_1$. Let $M = (C_1, C_2)$ where $C_i = (L_i, \Pi_i, br_i)$, L_i is the logic of logic programming under answer set semantics, $br_1 = \emptyset$ and $br_2 = \{l \leftarrow c_1 : l \mid l \in S\}$. It is easy to see that X is an answer set of Π iff $X = X_1 \cup X_2$ such that (X_1, X_2) is an equilibrium of M. This means that the equilibrium of the MCS (C_1, C_2) can be computed by Algorithm 2 without backtracking, i.e., ASP-Prolog^{MCS} could provide an alternative platform for the implementation of decomposable programs and iterative algorithms, as the above observation can be generalized to a splitting sequence of Π. We will show next that ASP-Prolog^{MCS} could be used to explore heuristics in answer set programming.

Let us consider the well-known graph coloring problem. Given a undirected graph (V, E), we would like to know whether the graph has a 3-coloring solution. The ASP encoding for computing a 3-coloring solution of a graph (V, E), denoted by $\Pi(V, E)$, is well-known and is omitted here to save space.

Table 1. ASP-Prolog^{MCS} in 3-coloring

Instance	colorable	ASP-Prolog^{MCS}			Instance	colorable	ASP-Prolog^{MCS}		
		Load	MCS	total			Load	MCS	total
p10000e10000	y	20.63	10.16	30.79	p10000e11000	y	22.75	12.56	35.31
p10000e12000	y	24.99	13.53	38.52	p10000e13000	y	25.74	17.87	43.61
p10000e14000	y	27.42	14.93	42.35	p10000e15000	y	31.93	16.68	48.61
p10000e16000	y	37.37	20.93	58.30	p10000e17000	y	52.71	18.99	71.70
p10000e18000	y	50.73	12.51	63.24	p10000e19000	y	50.46	11.79	62.25
p10000e20000	y	53.57	10.80	64.37	p10000e21000	n	59.02	4.51	63.53

A well-known heuristic for solving the 3-coloring problem is as follows. Let the degree of a node X, denoted by $degree(X)$, be the number of edges that have one endpoint in X. Let $R(V, E) = (V', E')$ be the graph obtained from (V, E) by removing all vertices whose degree is less than 3 and all the edges to/from these vertices. It is easy to see that $\Pi(V, E)$ has an answer set iff $\Pi(V', E')$ does. This process can be repeated until the degree of all vertices in the graph is at least three (including the possibility of the graph becoming empty). This can be used to create a partition $(V_1, \ldots, V_n)$ of V, where V_i is the set of vertices with degree less than 3 in the graph consisting of the vertices $\bigcup_{j=1}^{i} V_j$ and the edges among them that belong to E. Intuitively, V_i is the set of nodes removed at the ith iteration of the process described above. Using this partition, we can define a MCS $M = (C_1, \ldots, C_n)$ where, for each i, **(i)** kb_i contains the rules

$\Pi(V_i, E_i)$, where E_i is the set of edges from E between nodes in V_i; **(ii)** br_i contains the following constraints: for each $(p, q) \in E$ such that $q \in V_i$ and $p \in V_k$ with $k < i$:

$$\leftarrow (k : color(p, C)), color(q, C).$$

The Prolog code for computing the above partition is straightforward. We experimented this idea using the instances of the graph coloring problems obtained from `assat.cs.ust.hk/Assat-2.0/coloring-2.0.html`. The results of this experiment are presented in Table 1. In this table, each instance is of the form p*nem*, where n is the number of vertices and m is the number of edges in the graph. All the reported problems are randomly generated. The results show the simplicity of adding heuristics in the process. The execution times are relatively high compared to optimized ASP solvers (e.g., CLASP), due to the relative cost of decomposing the graph into MCS, but show success where other systems (e.g., standard Prolog or CLP(FD)) would fail.

5 Conclusions

In this paper, we presented two applications of the ASP-Prolog system. The first application deals with a large class of logic programs that are computationally easy, yet sometimes unsolvable for current ASP solvers. We showed how answer sets of this type of programs can be computed using the ASP-Prolog platform. The experimental results with the use cases highlight the potential of ASP-Prolog as a viable platform for the use of ASP in practical problems.

The second application is a centralized MCS system, ASP-Prolog^{MCS}, built using the facilities provided by ASP-Prolog. We described the implementation of the computation of equilibria semantics and encouraging experimental results. The system implements various algorithms that are required for the computation of equilibria of MCS systems; these are made possible by the specific capabilities of ASP-Prolog. The system can be used in applications that can be formulated as MCSs. The experimental evaluation is promising. Nevertheless, it also highlighted a need for improving the performance of ASP-Prolog^{MCS}. We believe that this can be achieved via targeted optimizations. In particular, we propose to explore mechanisms to optimize backtracking among modules (e.g., through caching mechanisms) and communication. This will be one of our goals in the nearest future.

As another future work of both applications, we propose to explore the role that the specific capabilities of ASP-Prolog (e.g., constraint solving capabilities) can have.

References

1. Bairakdar, S.E.-D., Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: The DMCS Solver for Distributed Nonmonotonic Multi-Context Systems. In: Janhunen, T., Niemelä, I. (eds.) JELIA 2010. LNCS, vol. 6341, pp. 352–355. Springer, Heidelberg (2010)
2. Balduccini, M., Gelfond, M.: The Language ASP{f} with arithmetic expressions and consistency-restoring rules. CoRR, abs/1301.1387 (2013)
3. Brewka, G., Eiter, T.: Equilibria in Heterogeneous Nonmonotonic Multi-Context Systems. In: AAAI, pp. 385–390 (2007)

4. Castro, L., Swift, T., Warren, D.: XASP: Answer Set Programming with XSB and Smodels. SUNY Stony Brook (2002), xsb.sourceforge.net/packages/xasp.pdf
5. Citrigno, S., et al.: The DLV system: Model generator and application frontends. In: WLP, pp. 128–137 (1997)
6. Dao-Tran, M., et al.: Distributed nonmonotonic multi-context systems. In: KR. AAAI Press (2010)
7. Elkhatib, O., Pontelli, E., Son, T.C.: ASP-Prolog: A System for Reasoning about Answer Set Programs in Prolog. In: Jayaraman, B. (ed.) PADL 2004. LNCS, vol. 3057, pp. 148–162. Springer, Heidelberg (2004)
8. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: *clasp*: A conflict-driven answer set solver. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 260–265. Springer, Heidelberg (2007)
9. Gelfond, M., Gabaldon, A.: From functional specifications to logic programs. In: ILPS, pp. 355–370. MIT Press (1997)
10. Gelfond, M., Lifschitz, V.: Logic programs with classical negation. In: ICLP, pp. 579–597 (1990)
11. Lifschitz, V., Turner, H.: Splitting a logic program. In: ICLP, pp. 23–38. MIT Press (1994)
12. Maddison, W., Maddison, D.: MacClade 4: Analysis of Phylogeny and Character Evolution. Sinauer (2000)
13. Marek, V., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm, pp. 375–398. Springer (1999)
14. Nguyen, N.-H., Son, T.C., Pontelli, E., Sakama, C.: ASP-prolog for negotiation among dishonest agents. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 331–344. Springer, Heidelberg (2011)
15. Niemelä, I.: Logic programming with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence 25(3,4), 241–273 (1999)
16. Pontelli, E., et al.: ASP at Work: An ASP Implementation of PhyloWS. In: ICLP. LIPICs (2012)
17. Pontelli, E., Son, T.C., Nguyen, N.-H.: Combining Answer Set Programming and Prolog: the ASP-Prolog System. In: Balduccini, M., Son, T.C. (eds.) Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning. LNCS, vol. 6565, pp. 452–472. Springer, Heidelberg (2011)
18. Sanderson, M., Purvis, A., Henze, C.: Phylogenetic Supertrees: Assembling the Trees of Life. Trends Ecol. Evol. 13, 105–109 (1998)
19. Semple, C., Steel, M.: A Supertree Method for Rooted Trees. Di. Ap. Math. 105, 147–158 (2000)
20. Simons, P., Niemelä, N., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence 138(1-2), 181–234 (2002)
21. Son, T.C., Pontelli, E., Nguyen, N.-H.: Planning for multiagent using ASP-Prolog. In: Dix, J., Fisher, M., Novák, P. (eds.) CLIMA X. LNCS, vol. 6214, pp. 1–21. Springer, Heidelberg (2010)
22. Yokoo, M., et al.: The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. IEEE Transactions on Knowledge and Data Engineering 10(5), 673–685 (1998)
23. Yokoo, M., Hirayama, K.: Algorithms for Distributed Constraint Satisfaction: A Review. Autonomous Agents and Multi-Agent Systems 3(2), 185–207 (2000)