Comparing different graph representations of logic programs under the Answer Set semantics

Stefania Costantini
Dept. of Pure and Applied Maths.
Univ. of L'Aquila
via Vetoio, Loc. Coppito, I-67100 L'Aquila, Italy
stefcost@univaq.it

Introduction

Stable Logic Programming (SLP) (MarTru99), also called Answer Set Programming (ASP) (Lif99), is an emergent, alternative style of logic programming: each solution to a problem is represented by an answer set, and not by answer substitutions produced in response to a query.

A main open problem is that of defining programming methodologies for SLP. Up to now, all programs discussed in the literature have a common, simple structure, that guarantees the existence of stable models (or, equivalently, of answer sets) by construction. As soon as SLP will be more widely and practically applied, the need will most likely arise of writing programs with a more complicated structure, and of composing existing programs into larger ones. Guidelines should be provided to programmers and system developers, in order to write consistent programs, and to combine them.

Thus, program analysis tools should be made available, so as to define and check properties of programs. These tools must be based upon some kind of representation of logic programs. Graph representations are simple and understandable. Graph-theoretic structures are able to represent properties of programs. Known properties of graphs can help improve the understanding of structural properties of the programs themselves.

As it is well-known, the traditional Dependency Graph (DG) is not able to represent programs under the Answer Set semantics: in fact, programs which are different in syntax and semantics, have the same Dependency Graph.

In this paper we try a formalization of the features that a graph representation of logic programs should exhibit. On the basis of this formalization, we compare three graph representations: the DG, the EDG and the RG.

The Extended Dependency Graph (EDG) (BCDP99) has been proposed by the author of this paper (together with others). It is based on explicitly representing the cycles which are present in the program, and their connections.

The Rule Graph (RG) has been introduced in (Dim96TCS), based on similar principles. Many interesting properties of the Answer Set semantics have been characterized in terms of properties of Rule Graphs.

We show that, unfortunately, also the RG is ambiguous with respect to the answer set semantics, while the EDG is isomorphic to the program it represents. We argue that the reason of this drawback of the RG as a software engineering tool relies in the absence of a distinction between the different kinds of connections between cycles.

Finally, we suggest that properties of a program might be characterized (and checked) in terms of *admissible colorings* of the EDG.

Background definitions

In this paper we consider the language $DATALOG^{\neg}$ for deductive databases, which is more restricted than traditional logic programming (the reader may refer to (MarTru99) for a discussion). In the following, we will implicitly refer to the ground version of $DATALOG^{\neg}$ programs.

A logic program may contain negated atoms of the form $\neg a$. A rule ρ is defined as usual, and can be seen as composed of a conclusion $head(\rho)$, and a set of conditions $body(\rho)$. The latter can be divided into positive conditions $pos(\rho)$ each one of the form A, and negative conditions $neg(\rho)$, each one of the form *not* A. The literal A is either an atom a, or a negated atom $\neg a$.

The answer sets semantics (GelLif88; GelLif91) is a view of logic programs as sets of inference rules (more precisely, default inference rules). Alternatively, one can see a program as a set of constraints on the solution of a problem, where each answer set represents a solution compatible with the constraints expressed by the program. Consider the simple program $\{q \leftarrow \textit{not}\, p.\ p \leftarrow \textit{not}\, q.\}$. For instance, the first rule is read as "assuming that p is false, we can *conclude* that q is true." This program has two answer sets. In the first one, q is true while p is false; in the second one, p is true while q is false.

When all literals are positive, we speak in terms of stable models [1].

[1] In (GelLif91), a syntactic transformation to logic programs with *not* but without $\neg$ is defined. Thanks to this reduction, in this paper we consider the stable model semantics as the founda-

A subset M of the Herbrand base B_P of a $DATALOG^{\neg}$ program P is a stable model of P, if M coincides with the least model of the reduct P^M of P with respect to M. This reduct is obtained by deleting from P all rules containing a condition *not* a, for some a in M, and by deleting all negative conditions form the other rules. Stable models are minimal supported models, and form an antichain.

Unlike with other semantics, a program may have several stable models, or may have no stable model, because conclusions are included in a stable model only if they can be justified. The following program has no stable model:
$\{a \leftarrow \textit{not}\, b.\ b \leftarrow \textit{not}\, c.\ c \leftarrow \textit{not}\, a.\}$
The reason is that in every minimal model of this program there is a true atom that depends (in the program) on the negation of another true atom. For instance, in the minimal model $\{a, b\}$, atom a depends on the negation of b; then, truth of a is not supported, and hence the model is not stable. Whenever a program has no stable models, we will say that the program is *inconsistent*. Correspondingly, checking for consistency means checking for the existence of stable models.

In the rest of the paper, for the sake of simplicity, and without loss of generality, we consider (as it is customary in the literature, and like (Dim96TCS)), *negative logic programs*, i.e. logic programs with no positive conditions in rules.

In particular, since (Cos95) we consider *kernel programs,* i.e. negative logic programs which have been simplified with respect to the (three-valued) well–founded model (VanGRosSch90). In fact, atoms which are either true or false in the well–founded model of a program, have this truth value in every stable model of the program itself. As discussed in (Cos95), finding stable models accounts to analyzing the relationships among atoms which have truth value "undefined" in the well–founded model.

A kernel program Π is characterized by having: i) well-founded model $WFS(\Pi) = \langle \emptyset; \emptyset \rangle$; ii) no positive conditions in rules, i.e. for every rule ρ, $pos(\rho) = \emptyset$; iii) every atom which is in the body of a rule must also appear in the head of some rule (possibly the same one). In the following sections, by "logic program" we will implicitly refer to a kernel program.

For a negative logic program P (and in particular for a kernel program P), the *dependency graph* $DG(P)$ is a finite directed graph whose vertices are the atoms occurring in P. There is an edge (w, v) iff there is a rule in P with v in its head and *not* w occurring in its body. In the DG, we say that atom a depends on atom b if there is a path from b to a. Atom a depends *evenly* (resp. *oddly*) on b if the path is composed of an even (resp. odd) number of arcs. A program is *stratified* if no atom depends on itself, and *call–consistent* if no atom depends *oddly* on itself. Stratified and call–consistent programs are guaranteed to be consistent under the answer set semantics (stable models always exist).

tion of answer set semantics; then, sometimes we use the terms stable model also for answer sets.

Properties of graph representations

As it is well–known, the Dependency Graph is not an adequate representation for logic programs under the stable model semantics, as shown by the following example.

Example 1 *Consider the following programs,* Π_1, Π_2 *and* Π_3*:*

$p \leftarrow \textit{not}\, p, \textit{not}\, e.$	$p \leftarrow \textit{not}\, p.$	$p \leftarrow \textit{not}\, p, \textit{not}\, e.$
$a \leftarrow \textit{not}\, b.$	$p \leftarrow \textit{not}\, e.$	$a \leftarrow \textit{not}\, b.$
$b \leftarrow \textit{not}\, a.$	$a \leftarrow \textit{not}\, b.$	$b \leftarrow \textit{not}\, a.$
$e \leftarrow \textit{not}\, f.$	$b \leftarrow \textit{not}\, a.$	$e \leftarrow \textit{not}\, f.$
$f \leftarrow \textit{not}\, h.$	$e \leftarrow \textit{not}\, f.$	$f \leftarrow \textit{not}\, h.$
$h \leftarrow \textit{not}\, e.$	$f \leftarrow \textit{not}\, h.$	$h \leftarrow \textit{not}\, e, \textit{not}\, a.$
$h \leftarrow \textit{not}\, a.$	$h \leftarrow \textit{not}\, e, \textit{not}\, a.$	

It is easy to see that the dependency graphs of the three programs coincide. Namely, the DG of the three programs is the leftmost graph of Figure1. However, Π_1 *has the stable model* $\{b, h, e\}$ *while instead* Π_2 *has the stable model* $\{a, f, p\}$ *and* Π_3 *has no stable models at all, i.e. it is inconsistent. Inconsistency is due to the presence of odd cycles: in fact, call-consistent programs, which do not contain odd-cycles, are always consistent. In the* DG *we can see two odd cycles: the first one involving atom p, which depends on itself; the second one involving atoms e,* f *and h.*

In the rest of the paper, we will introduce and discuss other graph representations, that appear more adequate than the DG.

In this section we state which are, in our view, the useful properties of any graph representation of logic programs.

Since we are interested in the syntactic structure of programs, let us characterize those programs that have the same structure, but different names for the atoms occurring in them.

Definition 1 *Let* P *be a logic program with Herbrand base* $B_P = \{a_1, a_2, \ldots, a_h\}$*, and let* B'_P *be a set of atoms with the same cardinality as* B_P*. A renaming* θ *is an injective and surjective function from* B_P *to* B'_P*, that can be denoted as a set* $\{(a_1, a'_1), (a_2, a'_2), \ldots, (a_h, a'_h)\}$ *where* $a_i \neq a_j$ *implies* $a'_i \neq a'_j$*, while for some* k *we may have* $a_k = a'_k$*.* θ *can be applied to* P*, thus obtaining a new program* $P' = P\theta$*, by replacing every occurrence of* a_i *by* a'_i*.*

Definition 2 *Two logic programs* P_1 *and* P_2 *are renaming–equivalent iff there exists a renaming* θ *from* B_{P_1} *to* B_{P_2} *such that* $P_2 = P_1\theta$*.*

Let $\mathcal{GF}$ be any graph representation formalism (for instance the DG). By $G \in \mathcal{GF}$ we mean a graph G which can be constructed for some program according to $\mathcal{GF}$. By $\mathcal{GF}(\mathcal{P})$ we mean the graph which represents program P according to $\mathcal{GF}$. The following is a straightforward property of any graph representation.

Definition 3 *A graph representation formalism* $\mathcal{GF}$ *is acceptable iff for every logic program* P*,* $\mathcal{GF}(\mathcal{P})$ *is unique.*

A stronger requirement is that, given a graph, we can reconstruct which is the program (or at least which is *a* program) that is represented by this graph. This requirement is related to the usefulness of the representation as a software engineering tool: given a program, one can determine how it is represented; given a graph, one can guess which kind of program it represents.

Definition 4 *A graph representation formalism $\mathcal{GF}$ is adequate iff for every graph $G \in \mathcal{GF}$, it is possible to construct from G a logic program P such that $G = \mathcal{GF}(\mathcal{P})$.*

We may expect that two programs which are renaming–equivalent are represented by graphs which have the same structure. Instead, it is reasonable to expect that different programs have different representations.

Definition 5 *A graph representation formalism $\mathcal{GF}$ is unambiguous iff for every two programs P_1 and P_2 which are not renaming–equivalent, $\mathcal{GF}(P_1) \neq \mathcal{GF}(P_2)$.*

For any graph representation formalism $\mathcal{GF}$ which is acceptable, adequate and unambiguous, we say that $\mathcal{GF}(\mathcal{P})$ is *isomorphic* to P. In fact, P can uniquely be determined from $\mathcal{GF}(\mathcal{P})$. More generally, we will say that the representation $\mathcal{GF}$ is isomorphic to logic programs. For short, we say that $\mathcal{GF}$ is an *isomorphic graph representation.*

Isomorphic graph representations are in our opinion the best software engineering tools: by means of these representations, a program construction methodology is able to clearly relate syntactic properties of programs, and properties of corresponding graphs. There are properties of graphs that correspond nicely, as we will see, to semantical properties of programs. Therefore, isomorphic graph representation are a means of relating syntax and semantics of logic programs.

Clearly, the Dependency Graph representation formalism is acceptable and adequate, but it is not unambiguous. In the example above in fact, it is impossible to say from the graph which of the three programs is being represented. Consequently, on the basis of the graph it is impossible to characterize consistency under the stable model semantics: we do not know from the graph whether the program under consideration has stable models, and it is impossible to find these models.

The relationship between cycles and answer sets: Intuition

Consider again the programs in Example 1 Why do they have such a diverse semantics, even though they contain the same cycles? What the DG, being ambiguous, does not show, is the structure of the connections between cycles.

In fact, the problem with an odd cycle under the stable model semantics is that the corresponding fragment of the overall program, taken alone, is inconsistent. Consider for instance the odd cycle involving atoms e, f, h in the DG of Example 1. The corresponding program fragment is $\{e \leftarrow \textit{not } f, f \leftarrow \textit{not } h, h \leftarrow \textit{not } e\}$. None of the atoms can be either true or false without causing a contradiction (a true atom which depends on the negation of a true atom). If, for instance, we assume f to be false, then e should be concluded to be true (by means of the first rule); consequently, h should be concluded to be false (by means of the third rule, since no other rules for h are available); but, at this point, f should be concluded to be true (by means of the second rule), which is a contradiction.

The only potential way of getting rid of inconsistency is that of connecting cycles. In previous work (BCDP99) we have identified two kinds of connections. Let rule C of the form $\alpha \leftarrow \textit{not } \beta$ be part an odd cycle. I.e., on the DG the edge connecting α to β belongs to an odd cycle.

First kind of connection:
α has an additional rule, for instance of the form $\alpha \leftarrow \textit{not } \delta$. In any stable model, if δ is false then α is true. We call *not* δ an *OR handle* for the cycle to which C belongs. Whenever δ is false and *not* δ is true, we say that the OR handle is *active.*

Second kind of connection:
C has an additional condition, for instance C is of the form $\alpha \leftarrow \textit{not } \beta, \textit{not } \gamma$.
In any stable model, if γ is true then α is false. We call *not* γ an *AND handle* for the cycle to which C belongs. Whenever γ is true and *not* γ is false, we say that the AND handle is *active.*

An active handle gives a truth value to an atom involved in a cycle. Consequently, the truth value of all the other atoms of the cycle follows without contradictions. A necessary condition for consistency is that every odd cycle has at least one handle. This condition is not sufficient, since two handles can be in conflict, in the sense that they cannot be active simultaneously. For instance, take program fragment $\{v \leftarrow \textit{not } v, \textit{not } w, t \leftarrow \textit{not } t, t \leftarrow \textit{not } w\}$: there are two odd cycles, one involving v with AND handle *not* w, and the other one involving t with OR handle *not* w. Since the same condition *not* w occurs in both handles, they cannot be simultaneously active, and therefore the cycle, and the overall program, will be inconsistent.

Notice that, whenever we have an even cycle without handles (*unconstrained* even cycle) we can choose any of the two models of the corresponding program fragment in order to form a stable model of the overall program. For instance, for even cycle $a \leftarrow \textit{not } b, b \leftarrow \textit{not } a$ any of the two models $\{a\}, \{b\}$ can be selected.

The programs Π_1 and Π_2 of Example 1 are divided into cycles as follows, where *OC* and *EC* denote odd and even cycle, respectively, and conditions appearing either in square brackets or in braces correspond to different kinds of handles.

Consider first program Π_1.

$$\begin{array}{rl}
OC_1: & \{\ p \leftarrow \textit{not}\, p, [\textit{not}\, e]. \\
EC_1: & \left\{ \begin{array}{l} a \leftarrow \textit{not}\, b. \\ b \leftarrow \textit{not}\, a. \end{array} \right. \\
OC_2: & \left\{ \begin{array}{l} e \leftarrow \textit{not}\, f. \\ f \leftarrow \textit{not}\, h. \\ h \leftarrow \textit{not}\, e. \end{array} \right. \\
H_1: & \{\ h \leftarrow \{\textit{not}\, a\}.
\end{array}$$

Condition *not* a in H_1 is an OR handle for OC_2. This handle becomes active by selecting the model $\{b\}$ for the unconstrained even cycle EC_1. This handle causes h and e to be true. Then, the AND handle of OC_1 becomes active, making p false. Consequently, Π_1 has the stable model $\{b, h, e\}$.

Similar considerations can be made on Π_2, even though it has a different structure: condition *not* a in this case is an AND handle for OC_2 (while in Π_1 it is an OR handle, instead).

$$
\begin{array}{rl}
OC_1: & \{\ p \leftarrow \textit{not}\, p. \\
H_2: & \{\ p \leftarrow \{\textit{not}\, e.\} \\
EC_1: & \begin{cases} a \leftarrow \textit{not}\ b. \\ b \leftarrow \textit{not}\ a. \end{cases} \\
OC_2: & \begin{cases} e \leftarrow \textit{not}\ f. \\ f \leftarrow \textit{not}\ h. \\ h \leftarrow \textit{not}\ e, [\textit{not}\ a]. \end{cases}
\end{array}
$$

If *not* a is false, then g is forced to be false, and consequently OC_2 has the stable model $\{f\}$. This means that the OR handle *not* e of OC_1 is true, and thus p is true: therefore the contradiction $p \leftarrow \textit{not}\, p$, which could determine the inconsistency, is made harmless. Consequently, Π_2 has the stable model $\{a, f, p\}$.

Finally, consider program Π_3:

$$
\begin{array}{rl}
OC_1: & \{\ p \leftarrow \textit{not}\, p, [\textit{not}\, e]. \\
EC_1: & \begin{cases} a \leftarrow \textit{not}\ b. \\ b \leftarrow \textit{not}\ a. \end{cases} \\
OC_2: & \begin{cases} e \leftarrow \textit{not}\ f. \\ f \leftarrow \textit{not}\ h. \\ h \leftarrow \textit{not}\ e, [\textit{not}\ a]. \end{cases}
\end{array}
$$

For the AND handle of odd cycle OC_2 to be active, we must select model $\{a\}$ for EC_1. In this way however, we have h false, f true and e false: this menas, the (unique) AND handle of odd cycle OC_1 is not active, and therefore Π_3 has no stable models.

Introducing Extended Dependency Graphs

In order to reason more directly and more efficiently about cycles and handles, we introduce a new graph representation of programs, since as we have seen the usual DG is not adequate to this aim. On this graph, we should be able of: detecting by means of efficient algorithms the syntactic features of programs; reasoning about the existence and the number of stable models; computing them. This new graph is similar to the DG, except it is more accurate for negative dependencies, and thus has been called EDG (Extended Dependency Graph).

The definition is based upon distinguishing among rules defining the same atom, i.e, having the same head. To establish this distinction, we assign to each head an upper index, starting from 0, e.g., $\{a \leftarrow c, \textit{not}\ b.\ a \leftarrow \textit{not}\ d.\}$ becomes $\{a^{(0)} \leftarrow c^{(0)}, \textit{not}\ b^{(0)}.\ a^{(1)} \leftarrow \textit{not}\ d^{(0)}.\}$.

The main idea underlying the definition is to create, for any atom a, as many vertices in the graph as the rules with head a (labeled $a^{(0)}, a^{(1)}, a^{(2)}$ etc., or, equivalently, a, a', a'', etc.).

Definition 6 *(Extended dependency graph) (EDG)*

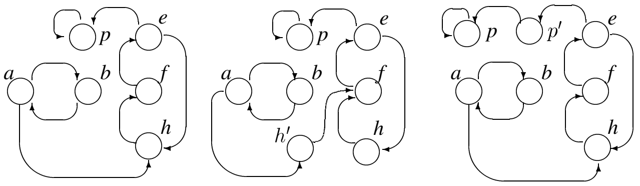

Figure 1: $EDG(\Pi_3)$ (left), $EDG(\Pi_1)$ (center) and $EDG(\Pi_2)$ (right). The leftmost graph coincides with the DG of the three programs.

For a logic program Π*, its associated Extended Dependency Graph* $EDG(\Pi)$ *is the directed finite graph* $\langle V, E\rangle$ *where the sets* V *and* E *are defined as follows.*

V: vertices *For each rule in* Π *there is a vertex* $a_i^{(k)}$*, where* a_i *is the name of the head and* k *is the index of the rule in the definition of* a_i*,*

E: edges *for each* $c_j^{(l)} \in V$*, there is an edge* $\langle c_j^{(l)}, a_i^{(k)}\rangle$*, if and only if not* c_j *appears as a condition in the k-th rule defining* a_i*.*

The definition of EDG extends that of DG in the sense that for programs where atoms are defined by at most one rule the two coincide. Consider in Figure 1 the EDG's of the programs in Example 1.

On the EDG's, we clearly see the cycles, and also the handles. In fact, rule $\{f \leftarrow \textit{not}\ h.\}$ *must* be represented by the two arcs $\langle h, f\rangle$ and $\langle h', f\rangle$, since truth of h may depend on any of its defining rules; the second one is auxiliary to the cycle, and corresponds to an OR handle. Therefore, the cycle has an OR handle *if and only if there is an incoming arc originated in a duplication of one of the atoms of the cycle*. In this case, the arc incoming into h' represents the OR handle of OC_2. In the same graph, arc $\langle e, p\rangle$ represents instead the AND handle of OC_1. Therefore, a cycle has an AND handle *if and only if* there exists an incoming arc into that cycle in the EDG, originated in (any duplication of) an atom not belonging to the cycle itself. *A cycle with no incoming arcs is unconstrained.*

The following result is easily stated.

Theorem 1 *The* EDG *is an isomorphic graph representation formalism.*

Consequently, the EDG conveys enough information for reasoning about stable models of the program. Apart of formal properties, from the point of view of the programmer, the EDG gives at first glance a hint on the program structure.

The Rule Graphs

The extended dependency graph introduced in this paper is not the only existing representation of negative logic programs.

A main contribution in this sense is the *Rule Graph* (RG) introduced in (Dim96TCS). If we let $r_1, \ldots, r_n$ be the rules of program Φ, the rule graph RG of Φ is a directed graph, with vertices corresponding to the r_i's, and there is and edge (r_1, r_2) whenever the conclusion of r_1 appears in the body of r_2.

The rule graph allows useful conditions about existence of stable models to be obtained, corresponding to formal properties coming from graph theory. In particular, stable models correspond to *kernels* of the RG.

With respect to the EDG, the rule graph RG has the same number of nodes, namely one per rule, and almost the same number of edges, where however the EDG in general has more edges. In particular, whenever there is only one atom in the body of each rule, the EDG and the RG are structurally identical, since the head of a rule being in the body of another collapses into the dependency between the two atoms. The EDG and RG may instead be different if there are more atoms in the body, case where the RG does not distinguish whether atoms in the body are one or more, and *which one* is the particular atom that is in common between two rules.

Below you find two different programs, which are not renaming–equivalent, that have different stable models, and different EDG's, but the same RG.

Example 2 *Let Φ_1 be:*

$r1.$ $a \leftarrow not\ b.$
$r2.$ $b \leftarrow not\ a.$
$r3.$ $g \leftarrow not\ g, not\ f.$
$r4.$ $f \leftarrow not\ q.$
$r5.$ $q \leftarrow not\ q, not\ a.$

The unique stable model of Φ_1 is $\{a, f\}$. In fact, not f is the unique AND handle for the odd cycle involving g. Then f must be true, and its truth is guaranteed by the truth of a, where not a is the unique AND handle of the odd cycle involving q. Consequently q is false.

Let Φ_2 be:

$r1.$ $a \leftarrow not\ b.$
$r2.$ $b \leftarrow not\ a.$
$r3.$ $g \leftarrow not\ g.$
$r4.$ $g \leftarrow not\ q.$
$r5.$ $q \leftarrow not\ q, not\ a.$

The unique stable model of Φ_2 is $\{a, g\}$. Again a must be true, since not a is the unique AND handle of the odd cycle involving q, which becomes false. In this program however, not q is the unique OR handle of the odd cycle involving g. The handle is thus active, and g is true.

The two programs have the same RG, shown in Figure 2.

In fact, in both cases we have that: the head of rule $r4$ is in the body of rule $r3$; the head of rule $r5$ is in the body of rule $r4$; atom a, which is the head of rule $r1$, is in the body of rule $r5$.

This means that:

(i) In our terminology, the RG is adequate only if a set of atoms is given, that is meant to constitute the Herbrand base of a program corresponding to the graph; it is otherwise impossible to guess a program from the RG.

(ii) the RG as a representation formalism is not unambiguous: in fact, it is impossible to distinguish between Φ_1 and Φ_2 on the basis of the rule graph;

(iii) consequently, the RG as a representation formalism is not isomorphic.

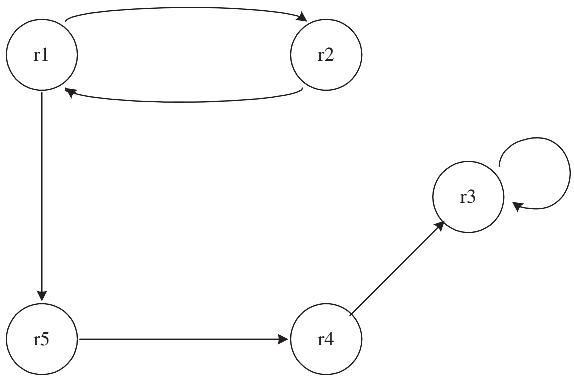

Figure 2: The Rule Graph of Φ_1 and Φ_2.

In conclusion, in our opinion it would be difficult to define any practical programming methodology on the basis of the rule graph, since it does not graphically distinguish among cases which are semantically very different.

Theorem 2 *Any graph representation formalism which is isomorphic to logic programs must necessarily distinguish between AND and OR handles of cycles.*

A new approach to study properties of programs

This section suggests how the EDG can be used to study the properties of logic programs under the Answer Set semantics, in terms of graph coloring. Let us define a *coloring* as an assignment of nodes of a graph to colors. Different graph colorings can be in principle defined, corresponding to specific properties of a program.

With respect to consistency, let the corresponding assignment be $\nu : V \to \{green,\ red\}$. An interpretation corresponds to a coloring, where all the true atoms are green, and all the others are red.

Below we specify which colorings we intend to rule out, since they trivially correspond to inconsistencies.

Definition 7 *(non-admissible coloring)*
Given logic program Π, a coloring $\nu : V \to \{green,\ red\}$ is non-admissible for $EDG(\Pi) = \langle V, E \rangle$ if and only if

1. $\exists i.\nu(v_i) = green$ *and* $\exists j.(v_i, v_j) \in E$ *and* $\nu(v_j) = green$, *or*
2. $\exists i.\nu(v_i) = red$ *and* $\forall j.(v_j, v_i) \in E$ *and* $\nu(v_j) = red$.

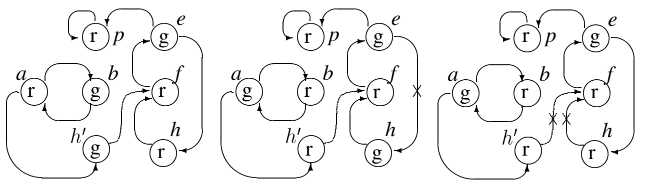

Figure 3: An admissible coloring of $EDG(\Pi_1)$ (on the left) and two not admissible ones (center and right, resp.) of $EDG(\Pi_1)$ with g=green, r=red and X=admissibility violation.

Essentially, green nodes cannot be adjacent and, moreover, edges to a red node cannot all come from red nodes.

A coloring for $EDG(\Pi)$ is admissible unless it is not admissible.

Example 3 *What are the admissible colorings for $EDG(\Pi_1)$ in Example 1?*

In the center coloring above, arc $\langle e, h\rangle$ violates admissibility. In fact, it corresponds to rule $h \leftarrow$ not e in Π_1. If e is true/green, then by the rule h cannot be concluded true/green. As a matter of fact, both e and h are true in the stable model of Π_1, but the truth of h comes from, intuitively, labeling h' green in the first coloring. In the right coloring above, admissibility is violated by arcs $\langle h', f, -\rangle$ and $\langle h, f, -\rangle$ which, together, represent the rule $f \leftarrow$ not h of Π_1. When all hs are red, we conclude h false and –by the above rule–, f true/green.

Now, we are able to state the relationship between admissible colorings and stable models of the given program.

Theorem 3 *An interpretation $\mathcal{I}$ is a stable model of Π if and only if it corresponds to an admissible coloring of $EDG(\Pi)$.*

Conclusions

Graph representations of logic programs can be important from a knowledge engineering perspective. In this paper, we have tried to make a step towards a classification of different representations. We have defined useful properties of representations, and on this ground we have compared (some of) the existing formalisms. A main topic of further research is that of showing that nice properties correspond to effective usefulness for program analysis and construction.

Acknowledgements

The EDG has been invented by the author of the present paper, together with Alessandro Provetti, Giampaolo Brignoli and Ottavio D'Antona. A main present research and implementation effort of this group focuses on stable models computation throughout EDG coloring. The idea is to exploit existing algorithms and heuristics for graph coloring, both for exact and approximate solutions.

References

Brignoli, G., 1998. *Extended Dependency Graphs for the analysis of logic programs,* Masters thesis (supervised by S. Costantini), Univ. of Milano, 1998.

Brignoli, G., Costantini, S. and Provetti, A., 1999. *A Graph Coloring algorithm for stable models generation,* Univ. of Milan Technical Report, submitted for publication.

Brignoli, G., Costantini, S., D'Antona, O. and Provetti, A., 1999. *Characterizing and computing stable models of logic programs: the non–stratified case,* Proc. of Conference on Information Technology, Bhubaneswar, India, December 1999.

Costantini, S., 1995. *Contributions to the stable model semantics of logic programs with negation,* Theoretical Computer Science, 149 (1995) : 231-255.

Cholewiński, P. and Truszczyński, M., 1996. *Extremal problems in logic programming and stable model computation.* Proc. of IJCSLP'96, pp. 408–422. Also in J. of Logic Programming, 38 (1999): 219–242.

Dimopoulos, Y. and Torres, A., 1996. *Graph theoretical structures in logic programs and default theories,* Theoretical Computer Science 170 (1996):209–244.

Dimopoulos, Y., Nebel, B. and Koehler, J., 1997. *Encoding planning problems in nonmonotonic logic programs,* European Workshop on Planning.

Eiter, T., Leone, N., Mateis, C., Pfeifer, G., and Scarcello, F., 1997. *A deductive system for non-monotonic reasoning,* Proc. Of the 4th LPNMR Conference, Springer Verlag, LNCS 1265: 363–374.

Gelfond, M. and Lifschitz, V., 1988. *The stable model semantics for logic programming,* Proc. of 5th ILPS conference: 1070–1080.

Gelfond, M. and Lifschitz, V., 1991. *Classical negation in logic programs and disjunctive databases,* New Generation Computing: 365–387.

Lifschitz V., 1999. *Answer Set Planning.* in: D. De Schreye (ed.) Proc. of the 1999 International Conference on Logic Programming (invited talk), The MIT Press: 23–37.

Marek, W., and Truszczyński, M., 1999. *Stable Models and an Alternative Logic Programming Paradigm,* In: The Logic Programming Paradigm: a 25-Year Perspective, Springer-Verlag: 375–398.

Subrahmanian, V.S., Nau D., and Vago C., 1995. *WFS + branch and bound = stable models,* IEEE Trans. on Knowledge and Data Engineering, 7(3):362–377.

Van Gelder A., Ross K.A. and Schlipf J., 1990. *The Well-Founded Semantics for General Logic Programs,* Journal of the ACM Vol. 38 N. 3.