Graph representations of logic programs: properties and comparison

Stefania Costantini[1] and Alessandro Provetti[2]

[1] Università di L'Aquila, Italy stefania.costantini@univaq.it
[2] Università di Messina, Italy ale@unime.it

Abstract. In this paper, we propose a formalization of the features that a graph representation of logic programs under the answer set semantics should in our opinion exhibit in order to be a satisfactory and useful representation formalism. We introduce a concept of isomorphism between a program and the corresponding graph. We show the importance of isomorphism for program analysis and we compare different graph representations w.r.t. isomorphism.

1 Introduction

Answer Set Programming (ASP) [1] [2] (also called Stable Logic Programming (SLP) [3]), is a relatively recent but well-established style of logic programming: each solution to a problem is represented by an answer set (also called stable model), and not by answer substitutions produced in response to a query.

With the growth of practical real-world ASP applications, the need is arising of writing programs with a complicated structure, and of composing existing programs into larger ones. Guidelines should be provided to programmers and system developers, in order to write consistent (or, sometimes, purposedly inconsistent) programs, and to combine them.

These guidelines should be based upon some kind of representation of logic programs. We believe that the properties of a good representation are at least the following: (i) the relationship between the program and its representation is clearly understandable; (ii) it is easy to shift back and forth from the program to the representation, so that a change in the program reflects into a change in the representation, and vice versa; (iii) it is possible to use the representation for developing techniques to define significant subprograms, study their interactions and determine which program parts can influence specific, critical points, and thus influence relevant program properties.

Graph representations are in our opinion good candidates. In fact, graphs 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.

In this paper we propose a formalization of the features that a graph representation of logic programs should in our opinion exhibit in order to be a satisfactory and useful representation formalism. We introduce a concept of isomorphism between a program and the corresponding graph, and we show that isomorphism is possible only if a graph

representation formalism is able to distinguish the cycles occurring in the program, and the different connections among them.

We discuss isomorphism w.r.t. three graph representations: the traditional Dependency Graph DG, the Extended Dependency Graph EDG [4] and the Rule Graph RG [5,6,7] (with some of its extensions). In a companion paper [8] we we show that isomorphic graphs can be used for characterizing properties of programs, first of all consistency, and (in perspective) for program debugging and composition.

The paper is organized as follows. In Section 2 we discuss properties of graph representations and we introduce isomorphic graph representations, and their relationship to cycles. In Section 3 we motivate these representations by showing how cycles occurring in a program affect the existence and number of answer sets. In Sections 4, 5 and 6 we examine various graph representations to establish their properties. Finally, Section 7 concludes the paper. The reader can find an introduction to ASP in Appendix A, and some basic definitions about graphs in Appendix B.

2 Properties of graph representations

The main property of programs one wants in the first place to study under the answer set semantics is consistency, i.e., the existence (and the number) of answer sets. In general, one has to compute and often examine the answer sets (e.g., in order to check whether some desired conclusion has been actually reached, or if a soft constraint is satisfied). In some situations, for instance for debugging a program or for trying to combine different program components, one may want to be able to investigate the program structure, in order to understand which parts influence which others, and then what ensures consistency, and what can spoil it [9]. When modifying and updating programs, it can be useful to reason about how the existence and content of answer sets is affected by modifications and/or updates performed on the given program.

As it is well-known, the Dependency Graph is not a satisfactory representation for logic programs under these respects, as shown by the following example.

Example 1. Consider the following programs Π_1, Π_2 and Π_3 (from left to right):

Π_1	Π_2	Π_3
$p \leftarrow not\, p, not\, e.$	$p \leftarrow not\, p.$	$p \leftarrow not\, p, not\, e.$
$a \leftarrow not\, b.$	$p \leftarrow not\, e.$	$a \leftarrow not\, b.$
$b \leftarrow not\, a.$	$a \leftarrow not\, b.$	$b \leftarrow not\, a.$
$e \leftarrow not\, f.$	$b \leftarrow not\, a.$	$e \leftarrow not\, f.$
$f \leftarrow not\, h.$	$e \leftarrow not\, f.$	$f \leftarrow not\, h.$
$h \leftarrow not\, e.$	$f \leftarrow not\, h.$	$h \leftarrow not\, e, not\, a.$
$h \leftarrow not\, a.$	$h \leftarrow not\, e, not\, a.$	

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 Figure 1. However, Π_1 has answer set $\{b, e, h\}$ while instead Π_2 has answer set $\{a, f, p\}$ and Π_3 has no answer sets at all, i.e. it is inconsistent.

Inconsistency may arise only because of odd cycles: in fact, call-consistent programs [10], which do not contain odd-cycles, are never inconsistent. In the DG of the

above programs 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.

However, the different connections between the odd cycles and the rest of the program, that cannot be seen on the DG, influence the existence and the number of answer sets.

Moreover, based on the DG it is impossible to make predictions about what happens in case of modifications. For instance, if adding fact e, the following happens: (i) program Π_1 remains consistent, with the two answer sets $\{b, e, h\}$ and $\{a, e, f\}$; (ii) Π_2 becomes inconsistent; (iii) Π_3 becomes consistent, with the two answer sets $\{b, e, f\}$ and $\{a, e, f\}$.

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 if 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 if for every logic program P, $\mathcal{GF}(P)$ is unique.*

A stronger requirement is that, given a graph, we can reconstruct the program that is represented by this graph, or at least *a* program, if there is more than one choice. 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 if 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. *An adequate graph representation formalism $\mathcal{GF}$ is unambiguous if for every two programs P_1 and P_2 which are not renaming-equivalent, we have that, apart from the labeling of vertices (if any), $\mathcal{GF}(P_1) \neq \mathcal{GF}(P_2)$.*

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.*

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 answer set semantics: we do not know from the graph whether the program under consideration has answer sets, and it is impossible to find these answer sets, and to predict whether they will change after updates.

3 The relationship between cycles and answer sets

There is a strong relationship between cycles occurring in a program, and existence and number of answer sets, as formally discussed in depth and at length in [9].

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 answer set 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 true (by means of the first rule); consequently, h should be concluded false (by means of the third rule, since no other rules for h are available); but, at this point, f should be concluded 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 [4] [9] we have identified (and formally defined) two kinds of connections. Let rule ρ of the form $\alpha \leftarrow \textit{not}\ \beta$ be part of a cycle C. I.e., on the DG the edge connecting β to α belongs to a cycle C.

First kind of connection:
the head α of rule ρ is defined also by (at least) another rule, for instance of the form $\alpha \leftarrow \textit{not}\ \delta$.
In any answer set, if δ is false then α is true. We call *not* δ an *OR handle* for the cycle to which ρ belongs. We also say that this OR handle is relative to atom α. Whenever δ is false and *not* δ is true, we say that the OR handle is *active.*

Second kind of connection:
rule ρ has an additional condition, for instance C is of the form
$\alpha \leftarrow \textit{not}\ \beta, \textit{not}\ \gamma$.
In any answer set, if γ is true then α is false. We call *not* γ an *AND handle* for the cycle to which ρ belongs. We also say that this AND handle is relative to atom α. 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 two handles of different kind, they cannot be simultaneously active, and therefore the cycle, and the overall program, will be inconsistent. Whenever we have an even cycle without handles (*unconstrained* even cycle) we can choose any of the two answer sets of the corresponding program fragment in order to form an answer set of the overall program. For instance, for even cycle $a \leftarrow \textit{not } b, b \leftarrow \textit{not } a$ any of the two answer sets $\{a\}, \{b\}$ can be selected.

In [8] we prove that a graph representation formalism $\mathcal{GF}$ which does not distinguish between AND and OR handles of cycles cannot be an isomorphic representation of logic programs.

4 The Extended Dependency Graph EDG

The Extended Dependency Graph (EDG) has been introduced in [4], where its properties were discussed for negative programs, i.e., program including only negative literals in the body of rules. These results are generalized to general programs in [8]. The EDG is similar to the DG, except it is more accurate for negative dependencies. 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 1, where upper index 0 is reserved to atoms with no defining rules. E.g., $\{a \leftarrow c, \textit{not } b.\ a \leftarrow \textit{not } d.\ b \leftarrow .\}$ becomes $\{a^{(1)} \leftarrow c^{(0)}, \textit{not } b^{(1)}.\ a^{(2)} \leftarrow \textit{not } d^{(0)}.\ b^{(1)} \leftarrow .\}$.

Definition 6. *(Extended dependency graph for general programs) (EDG)*
Given a general logic program Π, its associated Extended Dependency Graph $EDG(\Pi)$ is the directed finite labeled graph $\langle V, E, \{+, -\}\rangle$ where $\{+, -\}$ are the labels associated to the edges, and the sets V and E are defined as follows.

V:1 vertices *For each rule in Π there is a vertex $a_i^{(k)}$, $k \geq 1$, where a_i is the name of the head and k is the index of the rule in the definition of a_i.*

V:2 vertices *For each atom a_i in Π that does not occur in the head of any rule there is a vertex $a_i^{(0)}$.*

E:1 edges *For each $c_j^{(l)} \in V$, there is an edge $\langle c_j^{(l)}, a_i^{(k)}, +\rangle$, if and only if c_j appears as a positive condition in the k-th rule defining a_i.*

E:2 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 negative condition in the k-th rule defining a_i.*

The EDG for negative programs can be seen as a shortcut of the EDG for general programs, where each edge $\langle c_j^{(l)}, a_i^{(k)}\rangle$ is implicitly labeled with "$-$", i.e., corresponds to an edge $\langle c_j^{(l)}, a_i^{(k)}, -\rangle$.

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 EDGs of the programs in Example 1 (where $h^{(1)}$, $h^{(2)}$, $p^{(1)}$, $p^{(2)}$ are indicated as h, h', p, p' respectively, and all edges are implicitly labeled with '$-$').

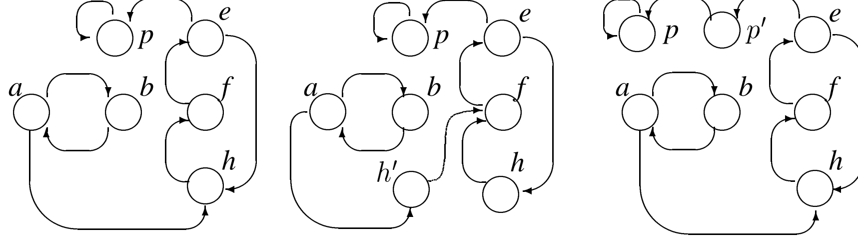

Fig. 1. $EDG(\Pi_3)$ (left), $EDG(\Pi_1)$ (center) and $EDG(\Pi_2)$ (right). The leftmost graph coincides with the DG of the three programs.

On the EDGs, we clearly see the cycles, and also the handles. Consider $EDG(\Pi_1)$ (center): rule $\{f \leftarrow$ *not* $h.\}$ is represented by the two edges $\langle h, f\rangle$ and $\langle h', f\rangle$, since truth of h may depend on any of its defining rules; the second rule for h is auxiliary to the cycle, and corresponds to an OR handle. Therefore, we can see on the EDG that the cycle has an OR handle because the edge incoming into h' originates in a duplication of one of the atoms of the cycle. In the same graph, edge $\langle e, p\rangle$ represents instead an AND handle. In fact, we can see on the EDG that the cycle has an AND handle because there is an incoming edge into that cycle, originated in a duplication of an atom not belonging to the cycle itself. The even cycle involving a and b has no incoming edges and then it is unconstrained.

In general: given program Π, a cycle in Π corresponds to a cycle in $EDG(\Pi)$. On the EDG: (1) An AND handle for a cycle C relative to atom α occurring in C is represented by an edge incoming into α (additional condition of the rule defining α). Atom α (and cycle C) may have either none or one or several AND handles. (2) An OR handle for a cycle C relative to atom α occurring in C is represented by a sibling node of α (additional rule defining α). More precisely, if C actually includes vertex α^i, another vertex α^j $j \neq i$ is also present in the EDG. Atom α (and cycle C) may have either none or one or several OR handles.

It is proved in [8] that the EDG is an isomorphic graph representation formalism.

5 The Rule Graph RG

An important graph representation of negative logic programs is the *Rule Graph* (RG) introduced in [5]. Given program Φ_g, assume first obtained a *reduced* program Φ, where rules with the same body that may occur in Φ_g are all merged into a single rule, whose head is the conjunction of the heads of the original rules. Let $r_1, \ldots, r_n$ be the rules

of the (reduced) program Φ, Then, 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 one of the conclusions of r_1 appears in the body of r_2.

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

Notice that merging rules with the same head is an essential feature of the approach, which leads to increased ambiguity with respect to the original program. Take for instance the program:

$l1.$ $a \leftarrow$ *not* b.
$l2$ $c \leftarrow$ *not* b.
$l3.$ $b \leftarrow$ *not* a.
$l4.$ $b \leftarrow$ *not* c.
$l5.$ $c \leftarrow$ *not* a.

The reduced version is

$r1.$ $a, c \leftarrow$ *not* b.
$r2.$ $b, c \leftarrow$ *not* a.
$r3.$ $b \leftarrow$ *not* c.

The kernels of the rule graph of the reduced version are two, namely $r1$ and $r3$. From the first one, going back to the reduced program one gets a stable model as the body of the corresponding rule, i.e., $\{b\}$. From the second one, again by taking the body of the corresponding rule, you get the stable model $\{c\}$. If, by abuse of notation, one draws the rule graph of the original program, no kernels are found.

For this reason, the RG and the EDG can be compared only on reduced programs, or, equivalently, on programs where all rules have different bodies. On this class of graphs, the RG and the EDG have the same structure, but with the important difference that vertices are labeled not with the name of the rules, but with (instances of) atoms. In particular, whenever there is only one atom in the body of each rule, this does not make any real difference, since the head of a rule being in the body of another collapses into the dependency between the two atoms. The difference instead exists and has a relevance whenever there are more atoms in the body of rules: in this case, 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. Then, the RG cannot distinguish between AND and OR handles, and consequently it cannot be an isomorphic graph representation formalism.

In fact, below we exhibit two different programs, which are not renaming-equivalent, and have both different answer sets and different EDGs, but the same RG.

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 answer set 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 answer set 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.

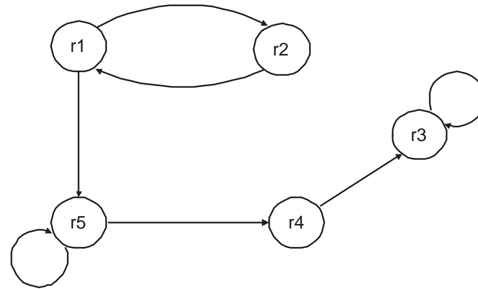

Fig. 2. The Rule Graph of Φ_1 and Φ_2.

This is because 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 $r5$.

We can instantiate the general argument performed in the proof of Theorem 1 for the RG.

(i) The RG is an adequate representation formalism if one guesses the Herbrand base of a program corresponding to the graph; many guesses are possible, the simplest one being to assign a different atom as the head of each different rule; other guesses are of course possible, where different rules have the same head.
(ii) as a consequence of not being able to tell the heads of the rules, 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; Since Φ_1 and Φ_2 are reduced programs, it is moreover impossible to get back to the original programs.
(iii) consequently, the RG is not an isomorphic graph representation formalism.

Then, in our opinion it would not be easy to define an interactive programming methodology on the basis of the rule graph, since it does not graphically distinguish among cases which are semantically very different.

This does not mean that the RG is not useful: on the contrary, it can be an effective program analysis tool. In fact, in [5] many examples are provided of useful program properties that can be identified on the RG, based on exploiting graph-theoretical properties.

6 Extensions to the Rule Graph

The Rule Graph RG has been extended in [6] to general logic programs, by introducing labeled edges so as to denote both positive and negative dependencies. Further extensions have then been introduced in [7]. They are based on identifying in a program P: the set of rules; the set $Head(P)$ of atoms occurring in the head of rules; the set $Body(P)$ of the conjunctions occurring as bodies in rules. In a body $B \in Body(P)$ one can distinguish the conjunction B^+ of positive literals and the conjunction B^- of negative literals.

- The *Body-Head Graph* has bodies and heads as vertices, and: an edge (B, h) from each body $B \in Body(P)$ and the head $h \in Head(P)$ of each rule where B occurs; a positive edge (h, B) from h to each body where it occurs positively; a negative edge (h, B) from h to each body where it occurs negatively.
- The *Rule-Head Graph* has rules and heads as vertices, and: an edge (r, h) from each rule $r \in P$ and its head h; a positive edge (h, r) from h to each rule where it occurs positively in the body; a negative edge (h, r) from h to each rule where it occurs negatively in the body.
- The *Body-Rule Graph* has bodies and rules as vertices, and: an edge (B, r) from each body $B \in Body(P)$ to each rule $r \in P$ whose body is B; a positive edge (r, B) from r to each body where the head h of r occurs positively; a negative edge (r, B) from r to each body where the head h of r occurs negatively.

Some combinations/variations of the above three representations are also defined in the above-mentioned papers.

These representations have been used as a basis for implementing the noMoRe answer set solver [11]. They have also been used in order to reason about loop formulas (positive cycles) [12].

However, none of them is isomorphic. In particular, they are acceptable and adequate but not unambiguous. In fact, atoms that are not defined (i.e., that do not occur in the head of any rule) are never considered in the definitions. Take for instance the programs $\{a \leftarrow b.\}$ and $\{a \leftarrow b, c.\}$. Indicate, e.g., the rule with r_1 and its body with B_1. These two programs have structurally identical graphs. We cannot distinguish between the two cases unless we explicitly associate the program to the graph (e.g., by copying rules and bodies into the vertices as labels).

7 Conclusions

Graph representations of logic programs can be useful from a software engineering perspective. We have defined which properties a graph representation should posses in

order to be a good program representation tool, and we have compared different representations w.r.t. these properties. Topics of further research are that of demonstrating effective usefulness of graph representations for program analysis and construction.

References

1. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In Kowalski, R., Bowen, K., eds.: Proc. of the 5th Intl. Conference and Symposium on Logic Programming, The MIT Press (1988) 1070–1080
2. Lifschitz, V.: Answer set planning. In: Proc. of the 16th Intl. Conference on Logic Programming. (1999) 23–37
3. Marek, V.W., Truszczyński, M. In: Stable logic programming - an alternative logic programming paradigm. Springer-Verlag, Berlin (1999) 375–398
4. Brignoli, G., Costantini, S., D'Antona, O., Provetti, A.: Characterizing and computing stable models of logic programs: the non–stratified case. In: Proceedings of the 1999 Conference on Information Technology, Bhubaneswar, India. (1999)
5. Dimopoulos, Y., Torres, A.: Graph theoretical structures in logic programs and default theories. Theoretical Computer Science **170** (1996) 209–244
6. Linke, T.: Graph theoretical characterization and computation of answer sets. In: Proceedings of IJCAI'01. (2001)
7. Konczak, K., Schaub, T., Linke, T.: Graphs and colorings for answer set programming: Abridged report. In Vos, M.D., Provetti, A., eds.: Answer Set Programming: Advances in Theory and Implementation, ASP03. Volume 78 of The CEUR Workshop Proceedings Series (2003) http://eur-ws.org/Vol-78/.
8. Costantini, S.: About graph representations of logic programs under the answer set semantics. Submitted, available from the author (2010)
9. Costantini, S.: On the existence of stable models of non-stratified logic programs. J. on Theory and Practice of Logic Programming **6**(1-2) (2006)
10. Fages, F.: Consistency of clark's completion and existence of stable models. Methods of Logic in Computer Science **2** 51–60
11. Anger, C., Konczak, K., Linke, T.: NoMoRe : A system for non-monotonic reasoning with logic programs under answer set semantics. Number 2083 in LNAI, Springer-Verlag, Berlin (2001) 325–330
12. Gebser, M., Schaub, T.: Loops: Relevant or redundant? Number 3662 in LNAI, Springer Verlag, Berlin (2005)
13. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing **9** (1991) 365–385
14. Costantini, S.: Contributions to the stable model semantics of logic programs with negation. Theoretical Computer Science **149** (1995) (prelim. version in Proc. of LPNMR93).
15. Costantini, S., Provetti, A.: Normal forms for answer sets programming. J. on Theory and Practice of Logic Programming **5**(6) (2005)
16. Berge, G.: Graphs and Hypergraphs. North Holland, Amsterdam (1973)

A Answer Set Programming

We consider the language $DATALOG^{\neg}$ for deductive databases, which is more restricted than traditional logic programming (the reader may refer to [3] for a discussion). In the following, we will implicitly refer to the ground version of $DATALOG^{\neg}$ programs.

A general logic program (or simply "logic program") is composed of rules 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 $\neg a$. A *negative* logic programs is composed of rules without positive conditions.

The answer sets semantics [1,13] 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 $\{q \leftarrow$ *not* $p.\ p \leftarrow$ *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.

A subset M of the Herbrand base B_P of a $DATALOG^{\neg}$ program P is an answer set 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. Answer sets are minimal supported models, and form an anti-chain.

Unlike with other semantics, a logic program may have several answer sets, or may have no answer set, because conclusions are included in an answer set only if they can be justified. The following program has no answer set:
$\{a \leftarrow$ *not* $b.\ b \leftarrow$ *not* $c.\ c \leftarrow$ *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. Whenever a program has no answer sets, we will say that the program is *inconsistent*. Correspondingly, checking for consistency means checking for the existence of answer sets.

Every logic program can be reduced to its *kernel normal form* [14] [15]. 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 [15] we have defined simple algorithms for shifting back and forth from logic programs to their kernel normal form.

In the following sections, by "logic program" we will implicitly refer to a general logic program. When explicitly indicated, we will refer to a negative program or to a kernel program.

B Graphs

The graph representations that we introduce in the following are all based on directed graphs. Then, in this paper the term "graph" refers implicitly to directed graphs. Below we introduce some basic definitions about graphs. For a comprehensive presentation and discussion on graphs and graph theory the reader may refer for instance to [16].

A *directed graph* or *graph* G is a pair $\langle V, E \rangle$. V is a set, whose elements are called *vertices* or *nodes.* The elements of E, where $E \subseteq V \times V$, are called the *edges* of the graph. For each edge $e \in E$, where $e = (v_i, v_j)$, $v_i, v_j \in V$, we say that e goes from

v_i to v_j, and, referring to v_j, that e is an edge coming from v_i. Let G be a graph, and $V' \subseteq V$. Let $V \setminus V'$ be the set of vertices that are are in V but not in V'.

A *labeled directed graph* or *labeled graph* G is a triple $\langle V, E, L\rangle$ where V is as before, L is a set of labels, and the elements of E, where $E \subseteq V \times V \times L$, are now of the form $e = (v_i, v_j, l)$, $v_i, v_j \in V$, $l \in L$. All definitions given here for directed graphs can be easily rephrased for labeled directed graphs, and then in the rest of the paper we often give this reformulation for granted.

Given a directed graph and vertices $a, b \in V$ (where it can be $a = b$) a *path* P from a to b is a subset E' of E where either $E' = (a, b)$ or there exists edge $(a, c) \in E'$ such that $E' \setminus (a, c)$ is a path from c to b. We say that P is *composed of* the edges (x, y) in E' and *involves* vertices x, y. By abuse of notation we also consider these vertices as belonging to the path, and we say $x, y \in P$. A *cycle* C is a set of edges which constitutes a path from v to v, for some $v \in V$.

A *simple* path from a to b is a path P such that there not exist $P' \subset P$ such that P' is a path from a to b. Vertex a is called the *source* of the path, while vertex b is called the *sink*. A *simple cycle* C is a set of edges which constitutes a simple path from v to v, for some $v \in V$. In the following, by "a cycle" we will implicitly mean a simple cycle. A cycle is *even* (resp. *odd*) if it is composed of an even (resp. odd) number of distinct edges.

A *kernel* in a directed graph is a subset V' of V such that (i) no two vertices in V' are joined by an edge in E (we say that V' is *independent*), and (ii) for every vertex $v \in V - V'$ there is some vertex $v' \in V'$ where (v', v) belongs to E (we say that V' is *dominating*).

Let E/V' be $E \cap (V' \times V')$. The *subgraph of* G *induced by* V' is the graph $G' = \langle V', E/V'\rangle$. If V' is a kernel, it is easy to see that (by condition (i) above) the induced subgraph has an empty set of edges.

For a negative logic program P (and in particular for a kernel program P), its *dependency graph* $DG(P)$ is a finite directed graph whose vertices are the atoms occurring in P. There is an edge (w, v) if and only if 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.

For a general logic program P, its *dependency graph* $DG(P)$ is a finite directed labeled graph whose vertices are the atoms occurring in P and whose set of labels is $\{+, -\}$. There is an edge $(w, v, +)$ if and only if there is a rule in P with v in its head and w occurring in its body (*positive* edge). There is an edge $(w, v, -)$ if and only if there is a rule in P with v in its head and *not* w occurring in its body (*negative* edge).

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 there exists a path between a and b composed of an even (resp. odd) number of arcs. A program is *stratified* if no atom depends on itself. As a special case one can have *positive dependencies*, that can be either even or odd, by considering (and thus counting) only the positive edges of the path. Similarly, one can consider *negative dependencies*, that can be either even or odd, by considering (and thus counting) only the negative edges of the path. A program is *stratified* if in its DG no atom depends on itself: i.e., for no atom a there is a cycle involving a.

A loop, or cycle, in a program corresponds to a cycle on its Dependency Graph $DG(P)$. A cycle composed of positive edges only is called a *positive cycle*. A cycle composed of negative edges only is called a *negative cycle*. Positive and negative cycles are either even or odd depending on the number of edges. A cycle which is not positive is said to be either even or odd according to the number of negative edges only, because consistency of the program is mainly affected by negative dependencies. A program is *call–consistent* if on its DG no atom has an odd negative dependency on itself: i.e., there is no cycle involving a and including an odd number of negative edges. Stratified and call–consistent programs are guaranteed to be consistent under the answer set semantics (answer sets always exist).