On the Equivalence and Range of Applicability of Graph-based Representations of Logic Programs

Stefania Costantini[a] * Ottavio D'Antona[b] and Alessandro Provetti[c]

[a]Dip. di Informatica, Università degli Studi di L'Aquila
Via Vetoio Loc. Coppito, L'Aquila, I-67100 Italy
stefcost@univaq.it
http://costantini.dm.univaq.it/

[b]Dip. di Scienze dell'Informazione, Università degli Studi di Milano
Via Comelico 39/41 Milan, I-20135 Italy
dantona@dsi.unimi.it
http://www.dsi.unimi.it/dantona/

[c]Dip. di Fisica, Università degli Studi di Messina
Salita Sperone 31 Messina, I-98166 Italy
ale@unime.it
http://www.unime.it/

Keywords. Logic Programming, Answer Set Programming, Graph Algorithms.

Logic programs under Answer Sets semantics can be studied, and actual computation can be carried out, by means of representing them by directed graphs. Several reductions of logic programs to directed graphs are now available. We compare our proposed representation, called Extended Dependency Graph, to the Block Graph representation recently defined by Linke [14]. On the relevant fragment of Well-founded irreducible programs, extended dependency and block graph turns out to be isomorphic. So, we argue that graph representation of general logic programs should be abandoned in favor of graph representation of well-founded irreducible programs, which are more concise, more uniform in structure while being equally as expressive.

1. Introduction

Answer Set Programming is a branch of Logic Programming based on the Stable Models and Answer Sets declarative semantics defined by Gelfond and Lifschitz [10, 11]. Essentially, the stable models semantics relates the negation-as failure operator *not* to the notion of consistent assumption in Reiter's default logic The Answer Sets semantics is an extension to the stable models semantics for programs that contain two kinds of negation: we will talk indifferently of answer sets or stable models, since this is not going to make a difference in the context of this letter.

*This work was partially supported by MIUR COFIN project *Aggregate- and number-reasoning for computing: from decision algorithms to constraint programming with multisets, sets, and maps.*

In standard logic programming program statements designate properties of a first-class object which is to be computed. Instead, Answer Sets Programming (ASP) is based on the understanding of program statements as constraints on a set of atoms that encode a solution to the problem at hand. Equivalently, autoepistemic logic supports the view of answer sets as coherent sets of beliefs that can be derived from themselves, in the sense that all conclusions in an answer set must be *supported* by other facts, that are seen as *hypotheses*, and that no true fact can be supported by the negation of another true fact (including, of course, itself).
For instance, consider a logic program composed of the following two rules:

$$
\begin{aligned}
p &\leftarrow \mathit{not}\ q\\
q &\leftarrow \mathit{not}\ p
\end{aligned}
$$

Conclusion p can be supported by hypothesis *not* q or, symmetrically, q can be supported by *not* p. In fact, this program has answer sets $\{p\}$ and $\{q\}$. If we read $\leftarrow$ as implication, and consider the two rules as a first-order theory, we have that the answer sets, in this case, coincide with the minimal models of this theory.

The one-rule program $r \leftarrow \mathit{not}\ r$ however, is contradictory, i.e. it has no answer set. In particular, the minimal model $\{r\}$ of the corresponding first-order theory is not an answer set, since atom r which is true in this model depends on its own negation. In general, the answer sets of a logic program are some of the minimal models of the corresponding first-order theory, and in particular they are those minimal models (if any) which are *supported*, in the sense that each atom true in the model can be derived by means of a rule whose conditions are true.

If we merge the two programs together, the resulting program

$$
\begin{aligned}
p &\leftarrow \mathit{not}\ q\\
q &\leftarrow \mathit{not}\ p\\
r &\leftarrow \mathit{not}\ r
\end{aligned}
$$

still has no answer set, since it contains a contradiction. If however we add for instance the rule $r \leftarrow \mathit{not}\ p$, we get a program with unique answer set $\{q, r\}$, where r is supported by *not* p, thus making the potential contradiction harmless. Alternatively, the inconsistency can be eliminated by modifying rule $r \leftarrow \mathit{not}\ r$ into rule $r \leftarrow \mathit{not}\ r, \mathit{not}\ q$, where assuming that q is true will force r to be false, thus obtaining the answer set $\{q\}$.

Then, a program may admit zero, one or more answer sets. Of particular interest is the problem of determining whether a program is consistent, i.e., it admits at least one answer set, which is also the basis for applying ASP to symbolic (logic-based) model checking [12].

Readers familiar with Prolog may refer to [13] for an introduction to ASP. For a comprehensive description of the ASP, the reader may refer to Marek and Truszczyński [15]. For an overview of the current research trends in ASP, the ASP workshop proceedings [18] could be taken as a guide.

In general, ASP is a very expressive and therefore complex logical formalism. If, as we do in this letter, we restrict ourselves to consider only programs without function symbols, then ASP solves all decision [19] and search [16] problems in NP. Therefore, it is important to provide computational mechanisms that are efficient on relevant subclasses of instances. Several implemented systems for answer sets computations are now available [20] and their performance is rapidly improving, approaching that of state-of-art SAT model checkers.

ASP solvers are employed as follows. Suppose that problem P at hand has solutions that can be described in terms of sets of symbols taken from a syntactic domain H (in our case, H is finite). The programmer shall write the logic program π, where each statement is seen as a constraint on the sets of symbols that are solution to the problem. By feeding π to an ASP solver, we obtain the answer sets; solutions to P can be *read off* the answer sets. The Herbrand base of π corresponds to or possibly extends H.

The basis for improving ASP solvers even further lies, in these authors' opinion, in a careful analysis of programs, so as to circumscribe as much as possible the sources of complexity, and in adoption of graphs as the main data structure, so as to reuse, directly, efficient graph-based algorithms.
In this perspective, we have contributed to Answer Set Programming by studying the syntactic structure of consistent programs [5, 7]; by defining concise normal forms [8] (called *kernel*), by proposing a graph-based representation of logic programs [3, 6] (called *Extended Dependency Graph*). The Extended Dependency Graph representation has enabled us to develop new answer set computation algorithms that are based on the finite- [17] and infinite-population [1] genetic models.

1.1. Recent Work Graph-based Representation of Logic Programs

In a recent IJCAI article, Linke [14] has proposed

i) a normal form for logic programs where rules have at most one positive condition, and

ii) a graph-based representation for such logic programs, called *block graph* (BG).

These two definitions come together in the definition of an answer set computation algorithm based on graph coloring. Linke compares his work with existing graph-based representations of logic programs of Dimopoulos and Torres [9] and ours; in [14] he says that

> [they] are *more or less* rule–based but have *serious drawbacks*: they deal only with prerequisite-free programs, because (w.r.t. Answer Sets semantics) there is some equivalent prerequisite-free program for each program. Since in general *equivalent prerequisite-free programs have exponential size of the original ones,* approaches which rely on this equivalence need exponential space. (our emphasis)

In this letter we study the relationship between Linke's block graphs and our Extended Dependency Graphs. Next, we point to some pitfalls in Linke's results about the connection between a-colorings of the Block graph and answer sets of the program considered. Finally, we address the claim that representations of prerequisite-free programs (i.e., no positive conditions) is subject to exponential blow-up of the instance size.

2. Technical preliminaries

This Section is intended for readers who are unfamiliar with Logic Programming. The standard definitions of (propositional) logic program syntax and those of Answer Sets [11] and Well-founded [21] semantics will be given.

Assume a language of constants and predicate constants. Assume also that terms and atoms are built as in the corresponding first-order language. Unlike classical logic and standard logic programming, no function symbols are allowed. A rule is an expression of the form:

$$\rho \,:\, A_0 \leftarrow A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n \quad (1)$$

where $A_0, \ldots A_n$ are atoms and *not* is a logical connective called *negation as failure*. Also, for every rule let us define $\mathit{head}(\rho) = A_0$, $\mathit{pos}(\rho) = A_1, \ldots, A_m$, $\mathit{neg}(\rho) = A_{m+1}, \ldots, A_n$ and $\mathit{body}(\rho) = \mathit{pos}(\rho) \cup \mathit{neg}(\rho)$. If $\mathit{body}(\rho) = \emptyset$ we refer to ρ as *a fact*, while if $\mathit{head}(\rho) = \emptyset$ we refer to ρ as *a constraint*, meaning that $\mathit{body}(\rho)$ *must be false in every answer set.*

A logic program is defined as a collection of rules. Rules with variables are taken as shorthand for the sets of all their ground instantiations and the set of all ground atoms in the language of a program Π will be denoted by $\mathbb{B}_\Pi$.

2.1. Semantics

For the sake of simplicity, we give here the definition of *stable model* instead of that of answer set, which is an extension given for programs that also contain the explicit negation operator $\neg$; this is not going to make a difference in the context of this work. Intuitively, a stable model is a possible view of the world that is *compatible* with the rules of the program. Rules are therefore seen as constraints on these views of the world.

Let us start by defining stable models of the subclass of positive programs, i.e. those where, for every rule ρ, $\mathit{neg}(\rho) = \emptyset$.

Definition 1 *(Stable Models of positive logic programs)*

The stable model $a(\Pi)$ *of a positive program* Π *is the smallest subset of* B_Π *such that for any rule (1) in* Π*:*

$$A_1, \ldots, A_m \in a(\Pi) \Rightarrow A_0 \in a(\Pi) \quad (2)$$

Clearly, positive programs have a unique stable model, which coincides with its minimal model, that can also be obtained applying other semantics; in other words positive programs are unambiguous. Moreover, the stable model of positive programs can be obtained as the fixpoint of the *immediate consequence operator* $T_\Pi(I) = \{A : \exists \rho \in \Pi \; s.t. \; A = head(\rho) \wedge \; pos(\rho) \subseteq I\}$. The iterated application of T_Π from $\emptyset$ on (i.e., $T_\Pi(\emptyset), T_\Pi^2(\emptyset), \ldots$) is guaranteed to have a fixed point, which corresponds to the answer set of Π.

A set of atoms S is a stable model of an (arbitrary) program if it is a minimal model, and it is supported. With respect of negation, we may notice that, if we assume S to be a stable model: (i) no atom can belong to S, which is derived by means of a rule with a condition *not* C where C is true in S, i.e. $C \in S$; (ii) all literals *not* B in the body of rules where B is false in S are, of course, true in S. Consequently, in order to check whether S actually is a stable model, all negations can be deleted according to the these criteria, in order to apply the above formulation for positive programs.

Definition 2 *(Stable Model of arbitrary logic programs)*

Let Π be a logic program. For any set S of atoms, let Π^S be a program obtained from Π by deleting

(i) *each rule that has a formula "not A" in its body with $A \in S$;*

(ii) *all formulae of the form "not A" in the bodies of the remaining rules.*

Since Π^S does not contain not , its stable model is already defined. If this stable model coincides with S, then we say that S is a stable model *of Π. Precisely, a stable model of Π is characterized by the equation:*

$$S = a(\Pi^S). \tag{3}$$

The Γ operator, introduced by Gelfond and Lifschitz in [10], is defined as $\Gamma(\Pi, S) = a(\Pi^S)$. When Π is fixed, we may drop the first parameter and refer to Γ as a function from the powerset of $\mathbb{B}_\Pi$ to itself. In practice however, stable models are not computed by applying Γ to all subsets of $\mathbb{B}_\Pi$. Answer set solvers [20] in fact apply more effective and smarter algorithms.

In this letter, consistency means existence of an answer set. Programs which have a unique stable model are called categorical. Entailment of atoms in the stable models semantics is defined in the standard way and it can be readily extended to arbitrary first-order formulae. However, it should be stressed that the real goal here is to compute solutions to the problem at hand in terms of sets of atoms (the answer sets) from which the solution is read out.

2.2. The Well-founded semantics

The Well-founded semantics of [21] assigns to a logic program Π a unique, three-valued model, denoted $WFS(\Pi) = \langle W^+, W^- \rangle$. Intuitively, W^+ is the set of atoms deemed true, W^- is the set of atoms deemed false, while atoms belonging to neither set are deemed *undefined.* Clearly, W^+ and W^- are disjoint; they can be defined as fixpoints of the Γ^2 operator, that is, Γ applied twice. Contrary to Γ, Γ^2 is monotonic, so it provably has at least one fixpoint. We define $W^+ = lfp(\Gamma^2)$, i.e., the least (smallest) $I \subseteq \mathbb{B}_\Pi$ such that $I = \Gamma^2(\pi, I)$. Conversely, we define $W^- = \mathbb{B}_\Pi \backslash gfp(\Gamma^2)$, i.e., W^- contains atoms that are not in the greatest fixpoint of Γ^2. Note that the least and greatest fixpoint can be derived from one another, in particular $gfp(\Gamma^2) = \Gamma(lfp(\Gamma^2))$. The sets W^+ and W^- are computed starting from what is certainly true (i.e. facts of the program) and what is certainly false, i.e. atoms not occurring in the head of any rule. Subsequent iterations extend these sets, until W^+ contains all atoms that are supported by a possible derivation, and W^- contains all atoms that have certainly no possible derivation. Atoms which are uncertain (basically, atoms directly or indirectly involved in negative circularities), are considered to be undefined.

For reasons that will be further discussed later on, in [8] we proposed to start computation of answer with a preliminary simplification of the program w.r.t. the well-founded semantics. In fact, it is well-known that answer sets are always a superset of the set of atoms which are true w.r.t. the well-founded model. In fact, answer sets always contain W^+ and are disjoint from W^-. The atoms that are relevant for deciding whether answer sets exist and finding them [5] are exactly those that are deemed undefined under the Well-founded semantics.

Definition 3 *A program* Π *is WFS-irreducible if and only if* $WFS(\Pi) = \langle \emptyset, \emptyset \rangle$.

That is, in WFS-irreducible programs all the atoms have truth value *undefined* under the Well-founded semantics.

Without loss of generality, we can always restrict to considering WFS-irreducible programs. Technically, the simplification of a program with respect to its well-founded model can be based on the application of the DBFZ algorithm of Dix et al. [2, 8]. For an arbitrary program Π, DBFZ produces a WFS-irreducible program $bdfz(\Pi)$ and a set of facts Φ. Each answer set of Π corresponds to an answer set of $bdfz(\Pi)$, union Φ.

The BDFZ algorithm has several attractive features: it runs in (low) polynomial time and performs loop detection, thus deleting from the program as many positive cycles as possible. The latter property is of interest for the following discussion.

3. Graph representation of logic programs

In literature, certain properties of logic programs have been studied throughout a graph representation called Dependency Graph (DG), where nodes represent atoms and arcs, labeled $+$ (resp. $-$) are representing a positive (resp. negative, i.e., using *not*) dependence between atoms. Dung (see [5]) provides a sufficient, but rather strong condition for consistency in terms of DGs. [3] and [6] show three programs that have the same DG but different semantics: one of them is inconsistent while the remaining two have answer sets different from each other. In other words, translating a program into its DG involves some loss of information, hence the DG does not seem an appropriate representation for developing answer set computation algorithms.

3.1. Extended Dependency Graphs

In [3] the EDG representation is introduced so as to achieve isomorphism (modulo the labeling of the nodes) between logic programs and directed graphs. The main shift in focus is that now vertices of the graph represent rules of the program, and are labeled with the atom in the conclusion of the rule itself. Modulo the labeling of nodes, these new graphs and programs are isomorphic [6], so it becomes possible to study consistency of program Π, and to compute its answer sets, in terms of $EDG(\Pi)$. This new graph is similar to the DG but it is more accurate in representing negative dependencies, and thus has been called EDG (Extended Dependency Graph). The definition of EDG extends that of DG in the sense that for programs where each atom is defined (appears as a conclusion) at most once, DG and EDG coincide.

The definition of EDG 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(for the sake of clarity, we may write a_i instead of $a_i^{(0)}$), e.g., $\{a \leftarrow c, not\ b.\ a \leftarrow not\ d.\}$ becomes $\{a^{(0)} \leftarrow c, not\ b.\ a^{(1)} \leftarrow not\ d.\}$. The main idea underlying the next definition is to create, for any atom a, as many vertices in the graph as the rules with head a (labelled $a, a^{(1)}, a^{(2)}$ etc.).

Definition 4 (from [3]) *(Extended dependency graph)*
For a logic program Π, *its associated Extended Dependency Graph* $EDG(\Pi)$ *is the directed finite labeled graph* $\langle V, E, \{+, -\}\rangle$ *defined below.*

V.1: *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;

V.2: *for each atom* u *never appearing in a head, there is a vertex simply labelled* u;

E.1: *for each* $c_j^{(l)} \in V$, *there is a positive 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, *and*

E.2: *for each* $c_j^{(l)} \in V$, *there is a negative edge* $\langle c_j^{(l)}, a_i^{(k)}, - \rangle$, *if and only if* c_j *appears as a negative condition in the k-th rule defining* a_i.

3.2. Computation of answer sets by EDG coloring

The EDG representation makes the design of an answer set computation algorithm conceptually simple. [8] proposes the following algorithm.

1. First, the program instance is syntactically simplified, reducing the instance to a negative program called the *kernel.* and a *store* (extensional database) of facts. The details of this reduction can be found in [8]. After applying BDFZ, for the sake of *time and space complexity* we perform further simplifications so as to get rid of positive atoms, that are irrelevant w.r.t. existence and number of answer sets. These simplifications are grounded in semantics considerations and preserve consistency and number of answer sets. As a result, the next phase needs to consider only EDGs described by cases V.1 and E.2 above.

2. Second, the EDG of the kernel version of the program is generated, and we start coloring, i.e., we search for distinguished 2-colorings, called *admissible*, of the EDG. Admissible colorings correspond one-to-one to answer sets of the kernel program, so it remains easy to extract the correspondent answer Set.

3. It also remains easy to project an answer set of the kernel to the corresponding answer sets of the original program, so there is a third, final phase where linear-time propagation algorithms are called to determine all atoms that are to be part of the extended answer set.

Notice that the central phase, the graph coloring, is the only one that implies search and in general NP-hardness. The initial and final phases take polynomial time.

3.3. Rule Graphs

The Rule Graph (RG) representation of Dimopoulos and Torres [9] allows useful conditions about existence of answer sets to be obtained, corresponding to graph-theoretical properties. Answer sets are characterized by the *kernels* of the *RG*.

Let $\{r_1, \ldots, r_n\}$ be the rules of a negative logic program Π, the rule graph $RG(\Pi)$ is a directed graph, with vertices corresponding to the r_i's, and there is and edge $\langle r_1, r_2\rangle$ whenever the conclusion of r_1 appears in the body of r_2.

The relationship between RGs and EDGs has been investigated by Costantini in [6]. 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. On the one hand, when all rules have only one condition in the body, 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. On the other hand, *EDG* and *RG* may differ when rules have more than one condition in the body; in such a case, we can say that the *RG* representation does not distinguish *which one condition* is in common between two rules [6].

While RGs are useful for studying program properties, they seem not immediately viable for the task of computing answer sets by graph algorithms.

3.4. Block Graphs

The Block Graph (BG) representation of logic programs is introduced by Linke for programs with at most one positive condition in each rule. Indeed, rules correspond to vertices of the graph while arcs represent the *block* relation between rules: rule ρ_1 blocks rule ρ_2 if an atom a is the conclusion of ρ_1 and a condition of ρ_2. Arcs are labeled 0 (resp. 1) depending on whether the conclusion in the origin rule appears as a positive (resp. negative) condition in ρ_2.
One important point to notice is that the graph of an arbitrary program Π is defined by first determining its *maximal grounded subset* Π', which, in short, is the greatest subprogram of Π *free* from

1. cyclic definitions through positive conditions (called positive cycles), and

2. undefined positive conditions.

In BG, arcs connect *only* vertices representing rules of Π'. Hence, the block graph thus results less dense than one would expect.

Example 1 *Let us consider program π_1:*

$r1.\quad a \leftarrow b.$
$r2.\quad b \leftarrow a.$
$r3.\quad c \leftarrow d.$
$r4.\quad e.$

π_1 is categorical, with answer set $\{e\}$; The maximum grounded subprogram is $\pi_1' = \{r4\}$. As a result, the block graph $BG(\pi_1)$ contains four nodes and no arcs. In fact, node for $r4$ has no incoming arcs since it is a fact. Nodes for $r1$, $r2$, and $r3$ have no incoming arcs since they do not belong to π_1'. So, all rules have neither 0-predecessors (positive conditions) nor 1-predecessors (negative conditions).

3.5. Computation of answer sets by BG coloring

Linke defines *a-coloring* of a BG, as special 2-coloring from vertices of the BG to $\{\oplus, \ominus\}$.

The a-coloring is related to answer sets by his Theorem 3.2, which states that i) a program Π has an answer set if and only if $BG(\Pi)$ admits an a-coloring and ii) the set of rules which are grounded w.r.t. an answer set A corresponds to the set of atoms that are in the conclusion of rules that have their node labeled $\oplus$ by the a-coloring. This suggests a straightforward extraction of A from the a-coloring. However, for programs that are not WFS-irreducible this characterization seems at least redundant.

Fact 1 *(counter-example to Th. 3.2 [14]) Consider again program π_1 from Example 1. This program has a positive cycle $\{a \leftarrow b, b \leftarrow a\}$, a rule, $r3$, with an undefined positive condition d and a rule, $r4$, which is a fact. Since $\{e\}$ is the unique answer set, it should correspond to an a-coloring where $r1$, $r2$, $r3$ are labeled $\ominus$ and $r4$ is labeled with $\oplus$.*

In contrast, if we consider Definition 3.1 (Block Graph) and Definition 3.2 (a-colorings) of [14], we conclude that all nodes are labeled $\oplus$.

In fact, the block graph of $BG(\pi_1)$ is composed of four isolated nodes, so by condition **A2** *of Definition 3.2, nodes with no 0-predecessors which have each 1-predecessor (if any) labeled $\ominus$ are labeled $\oplus$. Condition* **A2** *applies to all nodes of $BG(\pi_1)$, thus constituting a counterexample to Theorem 3.2: all rules should have their conditions satisfied but in actuality only $r4$ does.*

3.6. Possible repairs

The problem shown in Example 1 above would disappear if we restrict to considering the WFS-irreducible version of the program, which is equivalent to the original and equal to $\{r4\}$. In such a case, simplified program and maximal grounded program coincide and there are no isolated nodes. This change in Linke's algorithm *does not add computational costs* since computing the maximal grounded subset Π', which is required for building the block graph, is of complexity comparable to applying BDFZ.
In the end, the nature of this fix makes the case for applying graph representation only to Well-founded irreducible programs. Programs that are not WFS-irreducible can be efficiently reduced to WFS-irreducible form by applying BDFZ and dropping useless rules (or individual conditions) by an operator than simply extends Gelfond-Lifschitz Γ operator. Please refer to [8] for a discussion of this subject. It should be stressed that while this proposal comes from semantics analysis, it has an algorithmic value since BDFZ is an efficient (quadratic) algorithm.

To conclude with graph-theoretical considerations, we give the following equivalence result, which shows that on WFS-irreducible programs BG does not really extend EDG. Indeed, we can state the following.

Fact 2 *If Π is WFS-irreducible and devoid of positive cycles, then $EDG(\Pi)$ and $BG(\Pi)$ are isomorphic.*

Consider the following one-to-one correspondences from labels of EDG to labels of BG:

1. *a function λ that maps each node $a_i^{(k)}$ on the corresponding rule label, and*
2. *each arc $\langle a_i^{(k)}, b_j^{(l)}, -\rangle$ is mapped on $\langle \lambda(a_i^{(k)}), \lambda(b_j^{(l)}), 1\rangle$.*

4. Does reduction to Negative Programs Require Exponential Space?

The answer is *no,* and to support this claim we give below a simple, quadratic-time, linear-space transformation from WFS-irreducible programs to negative equivalent programs. Consider an arbitrary, WFS-irreducible program Π containing rules $\rho: \ a \leftarrow p_1, \ldots p_n, not\ n_l, \ldots not\ n_m$. Let us build the negative program Π^- in the following way.

1. Consider a positive-cycle-free version of the program; this version can be obtained either by applying BDFZ, or as follows. Take the positive version Π^+ of Π, i.e., drop all negative conditions. Build the EDG and identify all strongly-connected regions — corresponding to positive cycles. Remove from Π all rules involved in positive cycles. The resulting program is called $\Pi_{acyclic}$.
2. For each atom p_i that appears as a positive condition in $\Pi_{acyclic}$, add to Π^- the rule

   $\rho_{p_i}: \ p'_i \leftarrow not\ p_i$.
3. For each rule ρ of $\Pi_{acyclic}$ add to Π^- a corresponding rule ρ', with all positive conditions p_i substituted with conditions $not\ p'_i$.

Under WFS-irreducibility[2], the resulting program Π^- is equivalent to Π up to the language of the latter, i.e., primed atoms should be disregarded.

Theorem 1 *If* Π *is WFS-irreducible, then* Π *and* Π^- *are equivalent modulo projection over* $\mathcal{L}(\Pi)$*, the language of* Π.

The proof is by repeated application of Marek and Subrahmanian lemma. What is important in the transformation seen above it that Π^- is larger than Π only for the ρ_{p_i} rules. The total number of such rules has an upper bound in the number of atoms appearing in Π.

[2]For an example of a non-WFS-irreducible program for which step 2. of the transformation would not preserve equivalence, take program π_1 and check that under the transformation a spurious answer set $\{a, b, e\}$ appears.

5. Conclusions

In this letter we have argued about the usefulness of shifting from general logic programs to simplified well-founded irreducible programs, which are more concise and more uniform in structure while being equally expressive. This simplification is useful from a conceptual point of view: we have shown that is helps in evaluating and comparing different graph representations of logic programs. It is also useful from a practical point of view, since answer set solvers can perform the NP-hard search phase on a smaller instance.

In particular, we would like to point the reduction into the kernel normal form proposed in [8] (kernelization) as a *smart,* semantically-grounded transformation, that achieves very concise programs instances. One corroboration to this claim is that kernelization greatly simplify average user programs whereas *hard* programs, for instance Extremal Programs of [4], are already in kernel normal form.

Acknowledgments

Our work has benefitted from discussions with Pedro Cabalar, Michael Gelfond and Tomi Jahunen. Thanks also to David Lorenzo for having helped us with the literature.

REFERENCES

1. Bertoni A., Grossi G., Provetti A., Kreinovich V. and Tari L., 2001. *The Prospect for Answer Set Computation by a Genetic Model.* AAAI Spring Symposium ASP 2001, pp.1–5. AAAI press.
2. Brass S., Dix J., Freitag B., and Zukowski U., 2001. *Transformation-Based Bottom-Up Computation of the Well-Founded Model.* Theory and Practice of Logic Programming.
3. 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 1999 Conference on Information Technology, pp. 197–201.
4. 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.

5. Costantini, S., 1995. *Contribution to the stable model semantics of logic programs with negation.* Theoretical Computer Science, 149:231–255.
6. Costantini, S., 2001. *On the Existence of Stable Models of Non-stratified Logic Programs.* AAAI Spring 2001 Symposium *ASP 2001,* pp. 21–26. AAAI press, SS-01-01.
7. Costantini, S., 2002. *On the Existence of Stable Models of Non-stratified Logic Programs.* Submitted for pubblication.
8. Costantini, S., and Provetti A., 2002. *Normal Forms for Answer Set Programming.* Submitted for pubblication.
9. Dimopoulos, Y. and Torres, A., 1996. *Graph theoretical structures in logic programs and default theories,* Theoretical Computer Science 170:209–244.
10. Gelfond, M. and Lifschitz, V., 1988. *The stable model semantics for logic programming.* Proc. of 5th ILPS conference, pp. 1070–1080.
11. Gelfond, M. and Lifschitz, V., 1991. *Classical negation in logic programs and disjunctive databases.* New Generation Computing, pp. 365–387.
12. Heljanko, K. and Niemela, I., 2001. *Bounded LTL Model Checking with Stable Models.* Proc. of LPNMR01. Springer LNAI2173 pp. 200–212.
13. Lifschitz, V., 1999. *Answer Set Planning.* Invited talk. Proc. of ICLP '99 Conference, pp. 23–37. MIT Press.
14. Linke, T., 2001. *Graph Theoretical Characterization and Computation of Answer Sets,* In Proc. of IJCAI 2001.
15. 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.
16. Marek V. and Remmel J. , 2001. *On the expressibility of Stable Logic Programming.* AAAI Spring 2001 Symposium *ASP 2001,* pp. 124–131. AAAI press, SS-01-01.
17. Provetti A. and Tari L., 2000. *Answer Sets Computation by Genetic Algorithms, Preliminary Report.* Proc. of Genetic and Evolutionary Computation GECCO 2000 "Late breaking papers" track. pp. 303–308.
18. Provetti A. and Tran Cao S., editors, 2001. *Answer Set Programming: Toward Efficient and Scalable Knowledge Representation and Reasoning.* AAAI Spring Symposium. AAAI Press, SS-01-01
19. Schlipf J. , 1995. *The expressive powers of the logic programming semantics.* Journal of the Computer Systems and Science 51:64-86
20. Web location of the most known ASP solvers.
    CCALC:
    http://www.cs.utexas.edu/users/mcain/cc
    DeReS:
    http://www.cs.engr.uky.edu/~lpnmr/DeReS.html
    DLV:
    http://www.dbai.tuwien.ac.at/proj/dlv/
    NoMoRe:
    http://www.cs.uni-potsdam.de/~linke/nomore/
    SMODELS:
    http://www.tcs.hut.fi/Software/smodels/
21. 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.