Graph Theoretical Characterization and Computation of Answer Sets

Thomas Linke
Institut für Informatik, Universität Potsdam
Postfach 60 15 53, D–14415 Potsdam, Germany, linke@cs.uni-potsdam.de

Abstract

We give a graph theoretical characterization of answer sets of normal logic programs. We show that there is a one-to-one correspondence between answer sets and a special, non-standard graph coloring of so-called *block graphs* of logic programs. This leads us to an alternative implementation paradigm to compute answer sets, by computing non-standard graph colorings. Our approach is rule-based and not atom-based like most of the currently known methods. We present an implementation for computing answer sets which works on polynomial space.

1 Introduction

Answer set semantics [Gelfond and Lifschitz, 1991] was established as an alternative declarative semantics for logic programs. Originally, it was defined for extended logic programs[1] [Gelfond and Lifschitz, 1991] as a generalization of the stable model semantics [Gelfond and Lifschitz, 1988]. Currently there are various applications of answer set programming, e.g. [Dimopoulos *et al.*, 1997; Liu *et al.*, 1998; Niemelä, 1999]. Furthermore, there are reasonably efficient implementations available for computing answer sets, e.g. `smodels` [Niemelä and Simons, 1997] and `dlv` [Eiter *et al.*, 1997]. Systems, like `DeReS` [Cholewiński *et al.*, 1996] and `quip` [Egly *et al.*, 2000], are also able to compute answer sets, although they were designed to deal with more general formalisms.

Both systems and most of the theoretical results as well deal with answer sets in terms of atoms (or literals). This paper aims at a different point of view, namely characterizing and computing answer sets in terms of rules. Intuitively, the head p of a rule $p \leftarrow q_1, \ldots, q_n, not\ s_1, \ldots, not\ s_k$ is in some answer set A if $q_1, \ldots, q_n$ are in A and none of $s_1, \ldots, s_k$ is in A. Let

$$P = \left\{ \begin{array}{ccc} a \leftarrow b, not\ e & b \leftarrow d & c \leftarrow b \\ d \leftarrow & e \leftarrow d, not\ f & f \leftarrow a \end{array} \right\} \quad (1)$$

be a logic program and let us call the rules r_a, r_b, r_c, r_d, r_e, and r_f, respectively. Then P has two different answer sets $A_1 = \{d, b, c, a, f\}$ and $A_2 = \{d, b, c, e\}$. It is easy to see that the application of r_f blocks the application of r_e wrt A_1, because if r_f contributes to A_1, then $f \in A_1$ and thus r_e cannot be applied. Analogously, r_e blocks r_a wrt answer set A_2. This observation leads us to a strictly blockage-based approach. More precisely, we represent the block relation between rules as a so-called *block graph*. Answer sets then are characterized as special non-standard graph colorings of block graphs. Each node of the block graph (corresponding to some rule) is colored with one of two colors, representing application or non-application of the corresponding rule. The block graph has quadratic size of the corresponding logic program. Since the block graph serves as basic data structure for our implementation, it needs polynomial space.

[1] Extended logic programs are logic programs with classical negation.

2 Background

We deal with normal logic programs which contain the symbol *not* used for *negation as failure*. A *rule*, r, is any expression of the form

$$p \leftarrow q_1, \ldots, q_n, not\ s_1, \ldots, not\ s_k \quad (2)$$

where p, q_i $(0 \leq i \leq n)$ and s_j $(0 \leq j \leq k)$ are atoms. A rule is a *fact* if $n{=}k{=}0$, it is called *basic* if $k{=}0$. For a rule r we define $head(r){=}p$ and $body(r){=}\{q_1, \ldots, q_n, not\ s_1, \ldots, not\ s_k\}$. Furthermore, let $body^+(r){=}\{q_1, \ldots, q_n\}$ denote the positive part and $body^-(r){=}\{s_1, \ldots, s_k\}$ the negative part of $body(r)$. Definitions of the head, the body, the positive and negative body of a rule are generalized to sets of rules in the usual way. The elements of $body^+(r)$ are referred to as the *prerequisites* or *positive body atoms* of r. The elements of $body^-(r)$ are referred to as the *negative body atoms* of r. If $body^+(r){=}\emptyset$, then r is said to be *prerequisite-free*. A set of rules of the form (2) is called a *(normal) logic program*. A program is called *prerequisite-free* if each of its rules is prerequisite-free. We denote the set of all facts of program P by F_P.

Let r be a rule. r^+ then denotes the rule $head(r) \leftarrow body^+(r)$. For a logic program P let $P^+ = \{r^+ \mid r \in P\}$. A set of atoms X is *closed under* a basic program P iff for any $r \in P$, $head(r) \in X$ whenever $body(r) \subseteq X$. The smallest set of atoms which is closed under a basic program P is denoted by $\mathrm{Cn}(P)$.

The *reduct*, P^X, of a program P *relative to* a set X of atoms is defined by $P^X = \{r^+ \mid r \in P \text{ and } body^-(r) \cap X = \emptyset\}$. We say that a set X of atoms is an *answer set* of a program P iff $\mathrm{Cn}(P^X) = X$. The reduct P^X is often called the *Gelfond-Lifschitz reduction*. Observe, that there are programs, e.g. $\{p \leftarrow not\ p\}$, that do not possess an answer set. Throughout this paper, we use the term "answer set" instead of "stable

model" since it is the more general one.

A set of rules S of the form (2) is *grounded* iff there exists an enumeration $\langle r_i \rangle_{i \in I}$ of S such that for all $i \in I$ we have that $body^+(r_i) \subseteq head(\{r_1, \ldots, r_{i-1}\})$. For a set of rules S and a set of atoms X we define the set of generating rules of S wrt X as follows

$$GR(S,X) = \{r \in S \mid body^+(r) \subseteq X, body^-(r) \cap X = \emptyset\}. \quad (3)$$

The following result relates groundedness and generating rules to answer sets.

Lemma 2.1 *Let P be a logic program and X be a set of atoms. Then X is an answer set of P iff (i) $GR(P,X)$ is grounded and (ii) $X = \mathrm{Cn}(GR(P,X)^+)$.*

This lemma characterizes answer sets in terms of generating rules. Observe, that in general $GR(P,X)^+ \neq P^X$ (take $P = \{a \leftarrow, b \leftarrow c\}$ and $X = \{a\}$).

Assume that for each program P we have

$$\text{for each rule } r \in P \text{ we have } |body^+(r)| \leq 1. \quad (4)$$

In Section 5, we show how to generalize our approach to both normal logic programs with multiple positive body atoms and also to extended logic programs. Therefore, the assumption above is not a real restriction.

We need some graph theoretical terminology. A directed graph G is a pair $G = (V, A)$ such that V is a finite, non-empty set (vertices) and $A \subseteq V \times V$ is a set (arcs). For a directed graph $G = (V, A)$ and a vertex $v \in V$, we define the set of all *predecessors* of v as $\gamma^-(v) = \{u \mid (u, v) \in A\}$. Analogously, the set of all *successors* of v is defined as $\gamma^+(v) = \{u \mid (v, u) \in A\}$. A *path from v to v'* in $G = (V, A)$ is a finite subset $P_{vv'} \subseteq V$ such that $P_{vv'} = \{v_1, \ldots, v_n\}$, $v = v_1$, $v' = v_n$ and $(v_i, v_{i+1}) \in A$ for each $1 \leq i < n$. The arcs of a path $P_{vv'}$ are defined as $Arcs(P_{vv'}) = \{(v_i, v_{i+1}) \mid 1 \leq i < n\}$. A path from v to v for some $v \in V$ is called a *cycle* in G.

In order to represent more information in a directed graph, we need a special kind of labeled graphs. $(V, A^0 \cup A^1)$ is a directed graph whose arcs $A^0 \cup A^1$ are labeled with zero (*0-arcs*) and with one (*1-arcs*), respectively. For G we distinguish 0-predecessors (0-successors) from 1-predecessors (1-successors) denoted by $\gamma_0^-(v)$ ($\gamma_0^+(v)$) and $\gamma_1^-(v)$ ($\gamma_1^+(v)$) for $v \in V$, respectively. A path $P_{vv'}$ in G is called *0-path* if $Arcs(P_{vv'}) \subseteq A^0$. The *length* of a cycle in a graph $(V, A^0 \cup A^1)$ is the total number of 1-arcs occurring in the cycle. Additionally, we call a cycle *even* (*odd*) if its length is even (odd).

3 Block Graphs and Application Colorings

We now go on with a formal definition of the conditions under which a rule blocks another rule.

Definition 3.1 *Let P be a logic program s.t. condition (4) holds, and let $P' \subseteq P$ maximal grounded. The block graph $\Gamma_P = (V_P, A_P^0 \cup A_P^1)$ of P is a directed graph with vertices $V_P = P$ and two different kinds of arcs defined as follows*

$$A_P^0 = \{(r', r) \mid r', r \in P' \text{ and } head(r') \in body^+(r)\}$$
$$A_P^1 = \{(r', r) \mid r', r \in P' \text{ and } head(r') \in body^-(r)\}.$$

Observe, that there exists a unique maximal grounded set $P' \subseteq P$ for each program P, that is, Γ_P is well-defined. This definition captures the conditions under which a rule r' blocks another rule r (e.g. $(r', r) \in A^1$). We also gather all groundedness information in Γ_P, due to the restriction to rules in the maximal grounded part of logic program P. This is important because a block relation between two rules r' and r becomes effective only if r' is groundable through other rules. E.g. for program $P = \{p \leftarrow q, q \leftarrow p\}$ the maximal grounded subset of rules is empty and therefore Γ_P contains no 0-arcs. Figure 1 shows the block graph of program (1).

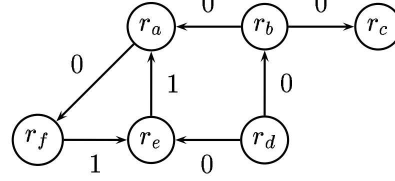

Figure 1: Block graph of program (1).

We now define so-called *application colorings* or *a-colorings* for block graphs. A subset of rules $G_r \subseteq P$ is a *grounded 0-path* for $r \in P$ if G_r is a 0-path from some fact to r in Γ_P.

Definition 3.2 *Let P be a logic program s.t. condition (4) holds, let $\Gamma_P = (P, A_P^0 \cup A_P^1)$ be the corresponding block graph and let $c : P \mapsto \{\ominus, \oplus\}$ be a mapping. Then c is an a-coloring of Γ_P iff the following conditions hold for each $r \in P$*

A1 *$c(r) = \ominus$ iff one of the following conditions holds*

a. $\gamma_0^-(r) \neq \emptyset$ and for each $r' \in \gamma_0^-(r)$ we have $c(r') = \ominus$

b. there is some $r'' \in \gamma_1^-(r)$ s.t. $c(r'') = \oplus$.

A2 *$c(r) = \oplus$ iff both of the following conditions hold*

a. $\gamma_0^-(r) = \emptyset$ or it exists grounded 0-path G_r s.t. $c(G_r) = \oplus$[2]

b. for each $r'' \in \gamma_1^-(r)$ we have $c(r'') = \ominus$.

Let c be an a-coloring of some block graph Γ_P. Rules are then intuitively applied wrt some answer set of P if they are colored $\oplus$. Condition **A1** specifies that a rule r is colored $\ominus$ (not applied) if and only if r is not "grounded" (**A1***a*) **or** r is blocked by some other rule (**A1***b*). A rule is colored $\oplus$ (applied) if and only if it is grounded (**A2***a*) **and** it is not blocked by some other rule (**A2***b*) . This captures the intuition which rules apply wrt to some answer set and which do not (see Section 1).

Let $r_p = p \leftarrow not\ p$. Then program $P = \{r_p\}$ has block graph $(P, \emptyset, \{(r_p, r_p)\})$. By Definition 3.2 there is no a-coloring of Γ_P. For this reason, we need both conditions **A1** and **A2**.

Lemma 3.1 *Let P be a logic program s.t. condition (4) holds and let c be an a-coloring of Γ_P. Then condition **A1** holds iff condition **A2** does not hold.*

In general, we do not have the equivalence stated in this lemma, because there are examples (see above) where no a-coloring exists. Lemma 3.1 states that a-colorings are well-defined in the sense that they assign exactly one color to each node.

[2] For a set of rules $S \subseteq P$ we write $c(S) = \oplus$ or $c(S) = \ominus$ if for each $r \in S$ we have $c(r) = \oplus$ or $c(r) = \ominus$, respectively.

We obtain the main result:

Theorem 3.2 *Let P be a logic program s.t. condition (4) holds and let Γ_P be the block graph of P. Then P has an answer set A iff Γ_P has an a-coloring c. Furthermore, we have $GR(P, A) = \{r \in P \mid c(r) = \oplus\}$.*

Answer sets can therefore be computed by computing a-colorings, e.g. $c(\{r_e\}) = \ominus$ and $c(\{r_d, r_b, r_c, r_a, r_f\}) = \oplus$ correspond to answer set A_1 of program (1).

4 Computation of A-colorings

For the following description of our algorithm to compute a-colorings, let P be some logic program s.t. condition (4) holds. Let $c : P \mapsto \{\ominus, \oplus\}$ be a partial mapping. c is represented by a pair of (disjoint) sets $(c_\ominus, c_\oplus)$ s.t. $c_\ominus = \{r \in P \mid c(r) = \ominus\}$ and $c_\oplus = \{r \in P \mid c(r) = \oplus\}$[3]. We refer to mapping c with the tuple $(c_\ominus, c_\oplus)$ and vice versa. Assume that Γ_P is a global parameter of each presented procedure (indicated through index P). Let U and N be sets of nodes s.t. U contains the currently uncolored nodes ($U = P \setminus (c_\ominus \cup c_\oplus)$) and N contains colored nodes whose color has to be propagated. Figure 2 shows the implementation of the non-deterministic procedure **color**$_P$ in pseudo code.

```
procedure  colorP(U, N : list; c : partial mapping)
 var  n : node;
  if  propagateP(N, c)  fails  then  fail;
  U := U \ (c⊖ ∪ c⊕);
  if  chooseP(U, c, n)  fails  then
     c := (c⊖ ∪ U, c⊕);
     if propagateP(U, c)  fails then fail  else  output  c;
  else
     U := U \ {n};
     c := (c⊖, c⊕ ∪ {n});
     if  colorP(U, {n}, c)  succeeds then exit
     else
        c := (c⊖ ∪ {n}, c⊕ \ {n});
        if colorP(U, {n}, c)  succeeds then exit  else  fail;
```

Figure 2: Definition of procedure **color**$_P$.

Notice that all presented procedures (except **choose**$_P$) return some partial mapping through parameter c or fail. **choose**$_P$ returns some node or fails.

When calling **color**$_P$ the first time, we start with $c = (\emptyset, F_P)$, $U = P \setminus F_P$ and $N = F_P$. That is, we start with all facts colored $\oplus$. Basically, **color**$_P$ takes both a partial mapping c and a set of uncolored nodes U and aims at coloring these nodes. This is done by choosing some uncolored node n ($n \in U$) with **choose**$_P$ and by trying to color it $\oplus$ first. In case of failure **color**$_P$ tries to color node n with $\ominus$. If this also fails **color**$_P$ fails. Therefore, we say that node n is used as a *choice point*. All different a-colorings are obtained by backtracking over choice points.

choose$_P(U, c, n)$ selects some uncolored node n ($n \in U$) s.t. $\gamma_0^-(n) = \emptyset$ and $\gamma_1^-(n) \neq \emptyset$ or the following condition holds:

CP there is some $n' \in \gamma_0^-(n)$ s.t. $c(n') = \oplus$.

[3] Since c is not total we do not necessarily have $P = c_\ominus \cup c_\oplus$.

If there is no such n then **choose**$_P$ fails. This strategy to select choice points ensures that nodes $c_\oplus$ are grounded. Observe that, if **choose**$_P$ fails and $U \neq \emptyset$ then we have to color all nodes in U with $\ominus$, since they cannot be grounded through rules $c_\oplus$.

During recursive calls N contains the choice point of the former recursion level. The color of nodes N has to be propagated with **propagate**$_P$ (see Figure 3) for two reasons. First, when coloring nodes in N with color x ($x \in \{\ominus, \oplus\}$) it is not checked whether this is allowed (wrt the actual c). This check is done by **propagate**$_P$. This means, **color**$_P$ fails only during propagation (see Theorem 4.1). Second, propagating already colored nodes prunes the search space and thus reduces the necessary number of choices. Since choice points make up the exponential part of our problem, propagation becomes the essential part of our approach.

Currently, we propagate only in arc direction as it is sufficient for correctness and completeness of the algorithm. Therefore, we have to deal with four propagation cases: if a node is colored x ($x \in \{\ominus, \oplus\}$) then this color has to be propagated over 1- and over 0-arcs. Let $r', r \in P$ be nodes s.t. $(r', r) \in A^0 \cup A^1$ and assume that r' is already colored. Then we have to propagate this color to node r. For example, propagating $c(r') = \oplus$ over 1-arcs gives $c(r) = \ominus$. For reasons of correctness, we cannot propagate colors without any further tests. We have got the following result.

Theorem 4.1 *Let P be a logic program s.t. condition (4) holds, let Γ_P be the corresponding block graph and let $c : P \mapsto \{\ominus, \oplus\}$ be a mapping. If c is an a-coloring of Γ_P then for each $r' \in P$ if $r' \in F_P$ then $c(r') = \oplus$ and the following conditions hold:*

(A) for each $r \in \gamma_1^+(r')$ we have $c(r) = \ominus$ if $c(r') = \oplus$

(B) for each $r \in \gamma_1^+(r')$ we have $c(r) = \oplus$ if $c(r') = \ominus$ and for each $r'' \in \gamma_1^-(r) : c(r'') = \ominus$ and $[\gamma_0^-(r) = \emptyset$ or there is some $r'' \in \gamma_0^-(r) : c(r'') = \oplus]$

(C) for each $r \in \gamma_0^+(r')$ we have $c(r) = \oplus$ if $c(r') = \oplus$ and for each $r'' \in \gamma_1^-(r) : c(r'') = \ominus$

(D) for each $r \in \gamma_0^+(r')$ we have $c(r) = \ominus$ if $c(r') = \ominus$ and for each $r'' \in \gamma_0^-(r) : c(r'') = \ominus$.

According to Theorem 3.2, a rule contributes to some answer set A if it is colored $\oplus$. In case of (A) there is no further condition, because a node r has to be colored $\ominus$ if there is some 1-predecessor r'' of r which is colored $\oplus$ (take $r'' = r'$ in **A1**b Definition 3.2). In other words, r cannot be applied if it is blocked by some other applied rule. Intuitively, condition (B) says that r has to be applied if all of its 1-predecessors are colored $\ominus$ (r is not blocked) and one of its 0-predecessors is colored $\oplus$ ($body^+(r)$ is a consequence of applied rules or r is a fact). Condition (C) states that rule r is applied if it is "grounded" through one of its 0-predecessors and if it is not blocked by some other rule. The last condition postulates that rule r cannot be applied if $body^+(r)$ cannot be derived from other applied rules. Theorem 4.1 implies that a mapping c is no a-coloring if **propagate**$_P$ fails. Figure 3 shows the implementation of **propagate**$_P$. The purpose of **propagate**$_P$ is to

```
procedure  propagate_P(N : list; c : partial mapping,)
 var  n' : node ;
    while  N ≠ ∅  do
        select  n'  from  N ;
        if  (n' ∈ c⊕)  then
(A)         if  propA_P(n', c)  fails  then  fail ;
(C)         if  propC_P(n', c)  fails  then  fail ;
        else
(B)         if  propB_P(n', c)  fails  then  fail ;
(D)         if  propD_P(n', c)  fails  then  fail .
```

Figure 3: Definition of procedure $\mathbf{propagate}_P$.

apply the corresponding propagation cases, e.g. if $c(n') = \oplus$ then cases (A) and (C) have to be applied.

The four procedures used in $\mathbf{propagate}_P$ can be easily implemented. For example, $\mathbf{propB}_P$ is shown in Figure 4. First,

```
procedure  propB_P(n' : node;  c : partial mapping)
 var  n : node ;  S : set of nodes ;
 S := {n ∈ γ1+(n') | condition (B) holds for n} ;
  while  S ≠ ∅  do
    select  n  from  S ;
    if  n ∈ c⊖  then  fail ;
    if  n ∉ c⊕  then
       c := (c⊖, c⊕ ∪ {n}) ;
       propagate_P(n, c) .
```

Figure 4: Definition of procedure $\mathbf{propB}_P$.

it determines the set S of all 1-successors of n' s.t. condition (B) holds. Finally, it tests whether all nodes in S can be colored $\oplus$. If node $n \in S$ is currently uncolored it is colored $\oplus$ and its color is propagated. If $c(n) = \ominus$ then $\mathbf{propB}_P$ fails, otherwise n is already colored $\oplus$ and we go on with the next node from S. The procedures for the remaining propagation cases can be implemented analogously. Whenever some currently uncolored node is colored during propagation, this color is recursively propagated by calling $\mathbf{propagate}_P$.

For partial mapping $c : P \mapsto \{\ominus, \oplus\}$ we define the set of *corresponding answer sets* A_c as

$$A_c = \{X \mid X \text{ is answer set of } P \text{ and } c_\oplus \subseteq GR(P,X) \text{ and } c_\ominus \cap GR(P,X) = \emptyset\}.$$

If c is undefined for all nodes then A_c contains all answer sets of P. If c is a total mapping then A_c contains exactly one answer set of P (if c is an a-coloring). With this notation we formulate the following result:

Theorem 4.2 *Let P be a logic program s.t. condition (4) holds, let c and $c(r)$ be partial mappings. Then for each $r \in (c_\ominus \cup c_\oplus)$ we have if* $\mathbf{propagate}_P(\{r\}, c)$ *succeeds and $c(r)$ is the actual partial mapping after executing* $\mathbf{propagate}_P$ *then $A_c = A_{c(r)}$.*

This theorem states that $\mathbf{propagate}_P$ neither discards nor introduces answer sets corresponding to some partial mapping c. Hence, it justifies that only nodes used as choice points lead to different answer sets. Therefore backtracking is necessary only over choice points (see Figure 2).

Define $C_P = \{c \mid c \text{ is some output of } \mathbf{color}_P(P \setminus F_P, F_P, (\emptyset, F_P))\}$. With this notation, we obtain correctness and completeness of $\mathbf{color}_P$.

Theorem 4.3 *Let P be a logic program s.t. condition (4) holds, let Γ_P be its block graph, let $c : P \mapsto \{\ominus, \oplus\}$ be a mapping and let C_P be defined as above. Then c is an a-coloring of Γ_P iff $c \in C_P$.*

Let us demonstrate how $\mathbf{color}_P$ computes the a-colorings of the block graph of program (1) (see Figure 1). We invoke $\mathbf{color}_P(U, N, c)$ with $U = P \setminus \{r_d\}$, $N = \{r_d\}$ and $c = (\emptyset, \{r_d\})$. First, $\mathbf{propagate}_P(N, c)$ is executed. By propagating $c(r_d) = \oplus$ with case (C) we get $c(r_b) = \oplus$ and recursively $c(r_c) = \oplus$. This gives $c = (\emptyset, \{r_d, r_b, r_c\})$. After updating uncolored nodes we obtain $U = \{r_a, r_e, r_f\}$. Now $\mathbf{choose}_P(U, c, n)$ (n variable) is executed. For $\mathbf{choose}_P$ there are two possibilities to compute the next choice point s.t. **CP** hold, namely r_a and r_e. Assume $n = r_a$. After updating U we have $U = \{r_e, r_f\}$ and the first recursive call $\mathbf{color}_P(U, \{r_a\}, c)$ is executed where $c = (\emptyset, \{r_d, r_b, r_c, r_a\})$. Again color $\oplus$ of node r_a has to be propagated by executing $\mathbf{propagate}_P(\{r_a\}, (\emptyset, \{r_d, r_b, r_c, r_a\}))$. By using propagation case (C) for $c(r_a) = \oplus$ we obtain $c(r_f) = \oplus$. This color is recursively propagated using case (A), which gives $c(r_e) = \ominus$. This leads to $c = (\{r_e\}, \{r_d, r_b, r_c, r_a, r_f\})$. Since U becomes the empty set, $\mathbf{choose}_P$ fails and c is the first output. Invoking backtracking means that the last recursive call to $\mathbf{color}_P$ fails. Then $c = (\{r_a\}, \{r_d, r_b, r_c\})$ and $\mathbf{color}_P(U, \{r_a\}, c)$ is executed. By using case (D) for $c(r_a) = \ominus$ we obtain $c(r_f) = \ominus$ and thus $c(r_e) = \oplus$ with case (B). Hence the second solution is $c = (\{r_a, r_f\}, \{r_d, r_b, r_c, r_e\})$. Since there is no other choice point, we have no further solutions.

5 Generalizations

In this section, we discuss generalizations of the presented approach. First, we show how to apply our method to normal logic programs with multiple positive body literals. Let P be a normal program without restrictions. For each rule $r = p \leftarrow q_1, \ldots, q_n, not\ s_1, \ldots, not\ s_k$ we define

$$P_r = \left\{ \begin{array}{l} p \leftarrow q', not\ s_1, \ldots, not\ s_k \\ q' \leftarrow q'_1, \ldots, q'_n \end{array} \right\} \cup \{q'_i \leftarrow q_i \mid 1 \le i \le n\} \quad (5)$$

where $q', q'_1, \ldots, q'_n$ are new atoms not appearing in P. For a program P we set $P_N = \bigcup_{r \in P} P_r$. Hence, we have defined a local program transformation which has linear size of the original program. Each normal program is transformed into some program in which $body^-(r) = \emptyset$ for each rule r with $|body^+(r)| > 1$. That is why we may interpret P_N as some kind of normal form of P. The following lemma obviously holds:

Lemma 5.1 *Let P be a normal logic program and let A be a set of atoms. Then A is an answer set of P iff there exists an answer set A_N of P_N s.t. A and A_N contain exactly the same atoms out of the set of all atoms occurring in P.*

It is straightforward to extend the algorithm presented in Section 4 to normal programs P_N. Let us call all rules $r \in P_N$ with $|body^+(r)| > 1$ *AND*-nodes and all other rules *OR*-nodes. Observe, that on the one hand, we do not have to modify Definition 3.1 of block graphs for programs P_N. It stays as

it is. We just distinguish two different kinds of nodes in Γ_{P_N}. On the other hand, Definition 3.2 and procedures presented in Section 4 deal only with OR-nodes and consequently we have to extend it to AND-nodes. For AND-node r we know by definition that $\gamma_1^-(r) = \emptyset$ (see (5)). Therefore, r cannot be blocked and we do not have to consider cases $\mathbf{A1}b$ and $\mathbf{A2}b$ of Definition 3.2. In order to extend Definition 3.2, we require that the following conditions hold for each AND-node r (corresponding to conditions $\mathbf{A1}a$ and $\mathbf{A2}a$ of Definition 3.2 for OR-nodes):

A3 $c(r) = \ominus$ iff there is some $r' \in \gamma_0^-(r)$ s.t. $c(r') = \ominus$

A4 $c(r) = \oplus$ iff for each $r' \in \gamma_0^-(r)$ we have $c(r') = \oplus$.

According to (5), for AND-node r we have $\gamma_1^-(r) = \gamma_1^+(r) = \emptyset$, $|\gamma_0^+(r)| = 1$ and $\gamma_0^+(r) \cup \gamma_0^-(r)$ will never contain any AND-node. For this reason, we obtain only two new propagation cases, in which we propagate the color of OR-nodes over 0-arcs to AND-nodes. Let $r \in \gamma_0^+(r')$ be some AND-node and let $x \in \{\ominus, \oplus\}$ be the actual color of r' (OR-node). Then we have the following new cases

(C') for each $r \in \gamma_0^+(r')$ we have $c(r) = \oplus$ if $c(r') = \oplus$ and for each $r'' \in \gamma_0^-(r) : c(r'') = \oplus$

(D') for each $r \in \gamma_0^+(r')$ we have $c(r) = \ominus$ if $c(r') = \ominus$,

which can be easily integrated into **propagate**$_P$.

[Gelfond and Lifschitz, 1990] show that logic programs with classical negation are equivalent to normal logic programs when new atoms are introduced. With this technique our approach is also suitable for computing answer sets of general logic programs (with classical negation). Additionally, we may apply techniques presented in [Janhunen *et al.*, 2000] to handle disjunctive logic programs.

6 Related Work

Directed graphs are often associated with logic programs and used in theory and applications. Usually, the nodes of these graphs are the atoms of the programs, e.g. dependency graphs or TMS networks [Doyle, 1979][4]. These approaches use rules to define graphs on atoms whereas we have used atoms to define graphs on rules.

Other approaches [Dimopoulos and Torres, 1996; Brignoli *et al.*, 1999] are more or less rule-based but have some serious drawbacks: they deal only with prerequisite-free programs, because (wrt to answer set 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.

In fact, the block graph is a specialization of graphs defined on rules in [Papadimitriou and Sideri, 1994; Linke and Schaub, 2000] for default theories. Whereas in the case of default logic the aforementioned graphs are abstractions of the essential blocking information, here they contain all information necessary for computing answer sets. Although we focus on the practical usage of the block graph it may also be used as a tool for theoretical analysis of logic programs. For example, results presented in [Linke and Schaub, 2000] imply that a logic program without odd cycles always has some answer set.

Clearly, **color**$_P$ is, like `smodels`, a Davis-Putnam like procedure. Once again, the main difference is that **color**$_P$ determines answer sets in terms of generating rules whereas `smodels` and `dlv` construct answer sets in terms of literals. Using rules instead of atoms has the advantage that we have complete knowledge about which rule is responsible for some atom belonging to an answer set. Atom-based approaches additionally have to detect the responsible rule and ensure groundedness, because in general there may be several rules with the same head. We obtain groundedness of generating rules as a by-product of our strategy to select choice points with procedure **choose**$_P$.

7 Conclusion

The main contribution of this paper was the definition of the block graph Γ_P of a program P. As a theoretical tool, the block graph seems to be suitable for investigations of many concepts for logic programs, e.g. answer set semantics, well-founded semantics or query-answering. As a first result, we have described answer sets as a-colorings (a non-standard kind of graph colorings of Γ_P). This led us to an alternative algorithm to compute answer sets by computing a-colorings which needs polynomial space.

Finally, let us give first experimental results to demonstrate the practical usefulness of our algorithm. We have used two NP-complete problems proposed in [Cholewiński *et al.*, 1995]: the problem of finding a Hamiltonian path in a graph (**Ham**) and the independent set problem (**Ind**).

Concerning time, our first prolog implementation (development time 6 month) is not comparable with state of the art implementations. However, Theorem 4.2 suggests to compare the number of used choice points, because it reflects how an algorithm deals with the exponential part of a problem. Unfortunately, only `smodels` gives information about its choice points. For this reason, we have concentrated on comparing our approach with `smodels`. Results are given for finding

	K_7	K_8	K_9	K_{10}
`smodels`	4800	86364	1864470	45168575
`noMoRe`	15500	123406	1226934	12539358

Table 1: Number of choice points for **HAM**-problems.

all solutions of different instances of **Ham** and **Ind**. Table 1 shows results for some **Ham**-encodings of complete graphs K_n where n is the number of nodes[5]. Surprisingly, it turns out that our non-monotonic reasoning system (`noMoRe`) performs very well on this problem class. That is, with growing problem size we need less choice points than `smodels`. This can also be seen in Table 2 which shows the corresponding time measurements. For finding all Hamiltonian cycles of a K_{10} we need less time than the actual `smodels` version. To be fair, for **Ind**-problems of graphs Cir$_n$[6]we need twice

[4]Truth maintenance systems can be translated into logic programs [Brewka, 1991].

[5]In a complete graph each node is connected to each other node.

[6]A so-called circle graph Cir$_n$ has n nodes $\{v_1, \ldots, v_n\}$ and arcs $A = \{(v_i, v_{i+1}) \mid 1 \leq i \leq n\} \cup \{(v_n, v_1)\}$.

the choice points (and much more time) `smodels` needs, because we have not yet implemented backward-propagation. However, even with the same number of choice points `smodels` is faster than `noMoRe`, because `noMoRe` uses general backtracking of prolog, whereas `smodels` backtracking is highly specialized for computing answer sets. The same applies to `dlv`.

	Ham for K_n			**Ind** for Cir_n		
$n =$	8	9	10	40	50	60
`smodels`	54	1334	38550	8	219	4052
`dlv`	4	50	493	13	259	4594
`noMoRe`	198	2577	34775	38	640	11586

Table 2: Time measurements in seconds for **HAM**- and **IND**-problems on a SUN Ultra2 with two 300MHz Sparc processors.

For future work, we may also think of different improvements of our algorithm. First of all, we have to integrate backward propagation (propagating colors against arc direction), since this will definitely improve efficiency by further reducing the number of choice points. Second, we may try to pre-color not only facts but also some other nodes, e.g. each node n with $(n, n) \in A^1$ has to be colored $\ominus$. The block graph may also be used for other improvements. For example, it is possible to replace 0-paths without incoming and outgoing 1-arcs by only one 0-arc. Finally, we have to investigate different heuristics for procedure $\mathbf{choose}_P$ to select the next choice point.

The approach as presented in this paper has been implemented in ECLiPSe-Prolog [Aggoun *et al.*, 2000]. The current prototype is available at http://www.cs.uni-potsdam.de/~linke/nomore.

Acknowledgements

I would like to thank T. Schaub, Ph. Besnard, K. Wang, K. Konczak, Ch. Anger and S.M. Model for commenting on previous versions of this paper.

This work was partially supported by the German Science Foundation (DFG) within Project “Nichtmonotone Inferenzsysteme zur Verarbeitung konfligierender Regeln”.

References

[Aggoun *et al.*, 2000] A. Aggoun, D. Chan, P. Dufresne and other. Eclipse User Manual Release 5.0. ECLiPSe is available at http://www.icparc.ic.ac.uk/eclipse, 2000.

[Brewka, 1991] G. Brewka. *Nonmonotonic Reasoning: Logical Foundations of Commonsense*. Cambridge University Press, Cambridge, 1991.

[Brignoli *et al.*, 1999] G. Brignoli, S. Costantini, O. D’Antona, and A. Provetti. Characterizing and computing stable models of logic programs: the non-stratified case. In *Proc. of Conf. on Information Technology*, pp. 197–201, 1999.

[Cholewiński *et al.*, 1995] P. Cholewiński, V. Marek, A. Mikitiuk, and M. Truszczyński. Experimenting with nonmonotonic reasoning. In *Proc. of the Int. Conf. on Logic Programming*, pp. 267–281. MIT Press, 1995.

[Cholewiński *et al.*, 1996] P. Cholewiński, V. Marek, and M. Truszczyński. Default reasoning system DeReS. In *Proc. KR-1996*, pp. 518–528. Morgan Kaufmann Publishers, 1996.

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

[Dimopoulos *et al.*, 1997] Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in non-monotonic logic programs. Proc. of the 4th European Conf. on Planing, pp. 169–181, Toulouse, France, 1997. Springer Verlag.

[Doyle, 1979] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[Egly *et al.*, 2000] U. Egly, Th. Eiter, H. Tompits, and St. Woltran. Solving advanced reasoning tasks using quantified boolean formulae. Proc. AAAI-2000, pp. 417–422, 2000.

[Eiter *et al.*, 1997] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. A deductive system for nonmonotonic reasoning. In J. Dix, U. Furbach, and A. Nerode, editors, *LPNMR-1997*, pages 363–374. Springer Verlag, 1997.

[Gelfond and Lifschitz, 1988] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. of the Int. Conf. on Logic Programming*, 1988.

[Gelfond and Lifschitz, 1990] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Proc. of the Int. Conf. on Logic Programming*, pp. 579–597, 1990.

[Gelfond and Lifschitz, 1991] M. Gelfond and V. Lifschitz. Classical negation in logic programs and deductive databases. *New Generation Computing*, 9:365–385, 1991.

[Janhunen *et al.*, 2000] T. Janhunen, I. Niemelä, P. Simons, and J. You. Unfolding partiality and disjunctions in stable model semantics. In A.G. Cohn, F. Guinchiglia, and B.Selman, editors, *Proc. KR-2000*, pp. 411–419, 2000.

[Linke and Schaub, 2000] T. Linke and T. Schaub. Alternative foundations for Reiter’s default logic. *Artificial Intelligence*, 124:31–86, 2000.

[Liu *et al.*, 1998] X. Liu, C. Ramakrishnan, and S.A. Smolka. Fully local and efficient evaluation of alternating fixed points. Proc. of the 4th Int. Conf. on Tools and Algorithms for the Construction Analysis of Systems, pp. 5–19, Lisbon, Portugal, 1998. Springer Verlag.

[Niemelä and Simons, 1997] I. Niemelä and P. Simons. Smodels: An implementation of the stable model and well-founded semantics for normal logic programs. In J. Dix, U. Furbach, and A. Nerode, editors, *Proc. LPNMR-1997*, pp. 420–429. Springer, 1997.

[Niemelä, 1999] I. Niemelä. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.

[Papadimitriou and Sideri, 1994] C. Papadimitriou and M. Sideri. Default theories that always have extensions. *Artificial Intelligence*, 69:347–357, 1994.