More on noMoRe

Thomas Linke **Christian Anger** **Kathrin Konczak**

Universität Potsdam, Institut für Informatik
August-Bebel-Strasse 89, D- 14482 Potsdam
{linke,canger,konczak}@cs.uni-potsdam.de

Abstract

We focus on the efficient computation of answer sets for normal logic programs. We concentrate on a recently proposed rule-based method (implemented in the `noMoRe` system) for computing answer sets. We show how `noMoRe` and its underlying method can be improved tremendously by improving its deterministic consequences. With these improvements `noMoRe` is able to deal with problem classes it could not handle so far.

1 Introduction

Answer set programming (ASP) is a programming paradigm, which allows for solving problems in a compact and highly declarative way. The basic idea is to specify a given problem in a declarative language, e.g. normal logic programs[1], such that the different answer sets given by answer sets semantics [8] correspond to the different solutions of the initial problem [10]. As an example, consider the independent set problem, which is to determine if there exists a maximal (wrt set inclusion) independent subsets of nodes for a given graph. A subset $S \subseteq V$ of nodes of a graph $G=(V,E)$ is called *independent* if there are no edges between nodes in S. Let

$$P = \left\{ \begin{array}{l} in(a) \leftarrow not\ in(d), not\ in(b) \\ in(b) \leftarrow not\ in(a), not\ in(c) \\ in(c) \leftarrow not\ in(b), not\ in(d) \\ in(d) \leftarrow not\ in(c), not\ in(a) \end{array} \right\} \quad (1)$$

be a logic program and let us call the rules r_a, r_b, r_c and r_d, respectively. Then program (1) encodes the independent set problem for graph $G = (\{a,b,c,d\}, \{(a,b),(b,c),(c,d),(d,a)\})$. Program (1) has two answer sets $X_1 = \{in(a), in(c)\}$ and $X_2 = \{in(b), in(d)\}$ corresponding to the two independent sets of graph G.

Currently there are reasonably efficient implementations (e.g. `smodels` [15] and `dlv` [5]) available as well as interesting applications of answer set programming. (e.g. [3, 13, 14]). Since computation of answer sets is NP-complete for normal logic programs (and Σ_2^P-complete for disjunctive logic programs), most algorithms rely on methods similar to the Davis-Putnam algorithm for SAT. That is, they contain a non-deterministic part (making choices) and a part computing deterministic consequences for these choices. Whereas in [7] different heuristics are investigated in order to make the "right" choices, in this paper we improve the deterministic consequences of the recently proposed rule-based `noMoRe` system [1]. For all ASP systems relying on Davis-Putnam-like algorithms (that is, for all systems mentioned so far), non-deterministic choices and deterministic consequences determine the behavior of the resulting algorithm. In particular, we redefine propagation of so-called a-colorings as introduced in [11] such that we are able to include *backward propagation*. Furthermore, we introduce a technique called *jumping* to ensure complete backward propagation and give experimental results showing the influence of the presented techniques.

2 Background

We deal with normal logic programs which contain the symbol *not* used for *negation as failure*. A *normal logic program* is a set of rules of the form $p \leftarrow q_1, \ldots, q_n, not\ s_1, \ldots, not\ s_k$ where p, q_i $(0 \leq i \leq n)$ and s_j $(0 \leq j \leq k)$ are propositional atoms. A rule is a *fact* if $n=k=0$, it is called *basic* if $k=0$ and *quasi-fact* if $n=0$. For a rule r like above we define $head(r)=p$ and $body(r) = \{q_1, \ldots, q_n, not\ s_1, \ldots, not\ s_k\}$. Fur-

[1] The language of normal logic programs is not the only one suitable for ASP. Others are disjunctive logic programs, propositional logic or DATALOG with constraints [4].

thermore, let $body^+(r) = \{q_1, \ldots, q_n\}$ denote the set of positive body atoms and $body^-(r) = \{s_1, \ldots, s_k\}$ the set of negative body atoms. We denote the set of all facts of a program P by $Facts(P)$ and the set of all atoms of P by $Atoms(P)$.

Let r be a rule. r^+ then denotes the rule $head(r) \leftarrow body^+(r)$, obtained from r by deleting all negative body atoms in the body of 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 $\mathrm{Cn}(P^X) = X$.

A set of rules P is *grounded* iff there exists an enumeration $\langle r_i \rangle_{i \in I}$ of P such that for all $i \in I$ we have that $body^+(r_i) \subseteq head(\{r_1, \ldots, r_{i-1}\})$. Observe that there exists a unique maximal grounded set $P' \subseteq P$ for each program P.[2] For a set of rules P and a set of atoms X we define the set of generating rules of P wrt X as $GR(P, X) = \{r \in P \mid body^+(r) \subseteq X, body^-(r) \cap X = \emptyset\}$. Then X is an answer set of P iff we have $X = \mathrm{Cn}(GR(P, X)^+)$. This 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 $X = \{a\}$).

We need some graph theoretical terminology . A *directed graph* (or *digraph*) 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 digraph $G = (V, A)$ and a vertex $v \in V$, we define the set of all *predecessors* of v as $\mathrm{Pred}(v) = \{u \mid (u, v) \in A\}$. Analogously, the set of all *successors* of v is defined as $\mathrm{Succ}(v) = \{u \mid (v, u) \in A\}$. 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 $1 \leq i < n$. Let $G = (V, A)$ and $G' = (V', A')$ be digraphs. Then G' is a *subgraph* of G if $V' \subseteq V$ and $A' \subseteq A$. G' is an *induced subgraph* of G if G' is a subgraph of G s.t. for each $v, v' \in V'$ we have that $(v, v') \in A'$ iff $(v, v') \in A$.

In order to represent more information in a directed graph, we need a special kind of arc labeling. $G = (V, A^0 \cup A^1)$ is a directed graph whose arcs $A^0 \cup A^1$ are labeled zero (A^0) and one (A^1). We call arcs in A^0 and A^1 *0-arcs* and *1-arcs*, respectively. For G we distinguish 0-predecessors (0-successors) from 1-predecessors (1-successors) denoted by $\mathrm{Pred0}(v)$ ($\mathrm{Succ0}(v)$) and $\mathrm{Pred1}(v)$ ($\mathrm{Succ1}(v)$) for $v \in V$, respectively. A path $P_{vv'}$ in G is called *0-path* if $Arcs(P_{vv'}) \subseteq A^0$.

[2]Here we mean sets P' which are maximal wrt set inclusion and grounded.

Block Graphs for Normal Logic Programs

Next we summarize the central definitions of block graphs for logic programs and a-colorings of block graphs (cf. [11, 12]).

Definition 1 *Let P be a logic program and let P' be the maximal grounded subset of P. 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*

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

This definition captures the conditions under which a rule r' blocks another rule r (i.e. $(r', r) \in A^1$). We introduce an 1-arc (r', r) in Γ_P if $r' = (q \leftarrow \ldots)$ and $r = (\ldots \leftarrow \ldots, not\ q, \ldots)$. We also gather all groundedness information in Γ_P, because we only introduce a 0-arc (r', r) (between rules $r' = (q \leftarrow \ldots)$ and $r = (\ldots \leftarrow q, \ldots)$) if r and r' are in the maximal grounded subset of P.[3] Figure 1 shows the block graph of program (1). Observe, that operations $head(r)$, $body^+(r)$ and $body^-(r)$ (for $r \in P$) are operations on the block graph, since the nodes of Γ_P are the rules of logic program P.

In order to define so-called *application colorings* or *a-colorings* for block graphs we need the following definition.

Definition 2 *Let P be a logic program and let $\Gamma_P = (V_P, A_P^0 \cup A_P^1)$ the corresponding block graph. Furthermore, let $x \in Atoms(P)$ and let $G_x = (V_x, A_x)$ be a subgraph of Γ_P. Then G_x is a* grounded 0-graph for x in Γ_P *iff the following conditions hold:*

1. *G_x is an acyclic subgraph of Γ_P s.t. $A_x \subseteq A_P^0$*
2. *G_x contains a target node r_x s.t. $x = head(r_x)$ and from every other node there exists a 0-path to the target node*
3. *for each node $r \in V_x$ we have $body^+(r) = \emptyset$ or for each $q' \in body^+(r)$ there exists a node $r' \in V_x$ s.t. $q' = head(r')$ and $(r', r) \in A_x$.*

[3]Observe, that 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.

Observe, that the nodes of a grounded 0-graph are grounded according to definition (see Section 2). Furthermore, the different grounded 0-graphs for atom x in Γ_P correspond to the different classical "proofs" for x in P^+, ignoring the default negations of all rules.

Definition 3 *Let P be a logic program, let $\mathcal{C} : P \mapsto \{\ominus, \oplus\}$ be a total mapping*[4]. *We call r* grounded wrt Γ_P and $\mathcal{C}$ *iff for each $q \in body^+(r)$ there exist a grounded 0-graph $G_q = (V_q, A_q))$ for q in Γ_P s.t. $\mathcal{C}(V_q) = \oplus$.*

A rule r is called blocked wrt Γ_P and $\mathcal{C}$ *if there exists some $r' \in Pred1(r)$ s.t. $\mathcal{C}(r') = \oplus$*[5].

Now we are ready to define a-colorings.

Definition 4 *Let P be a logic program, let Γ_P be the corresponding block graph and let $\mathcal{C} : P \mapsto \{\ominus, \oplus\}$ be a total mapping. Then $\mathcal{C}$ is an* a-coloring *of Γ_P iff the following condition hold for each $r \in P$*

AP *$\mathcal{C}(r) = \oplus$ iff r is grounded and r is not blocked wrt Γ_P and $\mathcal{C}$.*

Let $\mathcal{C}$ be an a-coloring of some block graph Γ_P. Rules are then intuitively applied wrt some answer set of P if and only if they are colored $\oplus$, that is, condition **AP** captures the intuition of applying a rule wrt to some answer set. Similarly, the negation of condition **AP** (r is **not** grounded **or** r is blocked) captures the intuition when a rule is not applicable wrt to some answer set.

Observe, that there are programs for which the corresponding block graph has no a-coloring and thus no answer set. Let r_p be rule $p \leftarrow not\ p$. Then program $P = \{r_p\}$ has block graph $(P, \emptyset, \{(r_p, r_p)\})$, that is, Γ_P consists of a single 1-loop. By Definition 4 there is no a-coloring of Γ_P. If we color r_p with $\oplus$ we get a direct contradiction to **AP**, since then r_p is blocked. On the other hand, if we color r_p with $\ominus$ then r_p is trivially grounded and not blocked. Therefore r_p has to be colored $\oplus$ which again is a contradiction.

The main result in [11] states that Program P has an answer set X iff Γ_P has an a-coloring $\mathcal{C}$ s.t.

$$GR(P, X) = \{r \in P \mid \mathcal{C}(r) = \oplus\}.$$

This result constitutes a rule-based method to compute answer sets by computing a-colorings. In Figure 1 we have depicted the two a-colorings of the block graph of program (1) left and right from '/', respectively.

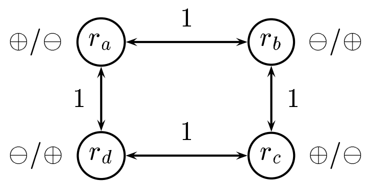

Figure 1: Block graph and a-colorings of program (1).

3 Propagation

In the **non**mono**to**nic **re**asoning system `noMoRe` [1] the approach described in the last section is implemented. Let us assume that each program is grounded. In order to describe the deterministic part of the implementation and its improvements, we need some central properties of nodes. All those properties are defined wrt partial a-colorings. We call a partial mapping $\mathcal{C} : P \mapsto \{\ominus, \oplus\}$ a *partial a-coloring* if $\mathcal{C}$ is an a-coloring of the induced subgraph of Γ_P with nodes $Dom(\mathcal{C})$[6].

Definition 5 *Let P be a logic program and let $\mathcal{C}$ be a partial a-coloring of Γ_P. For each node $r \in P$ we define the following properties wrt Γ_P and $\mathcal{C}$:*

1. p-grounded(r) *iff* $\forall q \in body^+(r) :$ $\exists r' \in Pred0(r) : q = head(r')$ *and* $\mathcal{C}(r') = \oplus$
2. p-notgrounded(r) *iff* $\exists q \in body^+(r) :$ $\forall r' \in Pred0(r) : q \neq head(r')$ *or* $\mathcal{C}(r') = \ominus$
3. p-blocked(r) *iff* $\exists r' \in Pred1(r) : \mathcal{C}(r') = \oplus$
4. p-notblocked(r) *iff* $\forall r' \in Pred1(r) : \mathcal{C}(r') = \ominus$.

Notice the difference between total and partial a-colorings. For example, if p-notgrounded(r) holds for r wrt some total a-coloring $\mathcal{C}$ then p-grounded(r) is the negation of p-notgrounded(r). This does not hold for partial a-coloring $\mathcal{C}$, since there may be nodes for which $\mathcal{C}$ is undefined. For this reason, we have to define both p-grounded (p-blocked) and p-notgrounded (p-notblocked), respectively, because they cannot be defined through each other wrt partial a-colorings. However, we have the following result for total a-colorings:

Theorem 1 *Let P be a logic program and let $\mathcal{C}$ be a total a-coloring of Γ_P. Then for each node $r \in P$ we have r is grounded wrt Γ_P and $\mathcal{C}$ iff p-grounded(r) wrt*

[4] A mapping $\mathcal{C} : P \mapsto C$ is called *total* iff for each node $r \in P$ there exists some $\mathcal{C}(r) \in C$. Oppositely, mapping $\mathcal{C}$ is called partial if there are some $r \in P$ for which $\mathcal{C}(r)$ is undefined.

[5] We say a rule is grounded or blocked (omitting $\mathcal{C}$) if it is clear from the context with respect to which mapping $\mathcal{C}$ the rule is grounded or blocked, respectively.

[6] $Dom(\mathcal{C})$ denotes the domain of mapping $\mathcal{C}$.

Γ_P and $\mathcal{C}$. Furthermore, we have r is blocked wrt Γ_P and $\mathcal{C}$ iff p-blocked(r) wrt Γ_P and $\mathcal{C}$.

Clearly, to be grounded (wrt Γ_P and $\mathcal{C}$ for node r) is a global concept wrt Γ_P whereas p-grounded(r) is defined locally wrt Γ_P. Furthermore, observe the difference between r is blocked and p-blocked(r) wrt Γ_P and $\mathcal{C}$. Even if the definitions of both concepts are the same (cf Definition 3), the former is defined wrt to total a-colorings, whereas the later one is defined wrt partial a-colorings. In a situation like in Figure 2 we do not have p-blocked(r') and we do not have p-notblocked(r') wrt the depicted partial a-coloring. But we always have either that r is blocked or not blocked wrt total a-colorings.

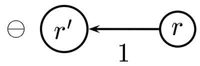

Figure 2: Some block graph with partial a-coloring.

Definition 6 *Let $\mathcal{C}$ be a partial a-coloring of Γ_P and let U be the set of uncolored nodes wrt $\mathcal{C}$. Then each node $r \in U$ can be* colored $\oplus$ by propagation of $\mathcal{C}$ *iff we have p-grounded(r) and p-notblocked(r) wrt $\mathcal{C}$. Node r can be* colored $\ominus$ by propagation of $\mathcal{C}$ *iff we have p-notgrounded(r) or p-blocked(r) wrt $\mathcal{C}$.*

Notice, that propagation of partial a-colorings to uncolored nodes is global wrt Γ_P, since in order to propagate $\mathcal{C}$ as much as possible we have to check all nodes in U which in general are distributed over Γ_P. According to Definitions 5 and 6 nodes colored by propagation always have colored predecessors. Therefore we obtain a more procedural way to propagate partial a-colorings by *localized* propagation conditions.

Definition 7 *Let P be a logic program, let Γ_P be the corresponding block graph and let $\mathcal{C}$ be a partial a-coloring of Γ_P. We define an extented mapping $\mathcal{C}^e$ of $\mathcal{C}$ s.t. for each $r \in Dom(\mathcal{C})$ we have $\mathcal{C}^e(r) = \mathcal{C}(r)$ and for each $r, r' \in P$ the following conditions hold wrt Γ_P and $\mathcal{C}^e$:*

(A) *if $r \in$ Succ1(r') and $\mathcal{C}^e(r') = \oplus$ then $\mathcal{C}^e(r) = \ominus$*

(B) *if $r \in$ Succ1(r') and $\mathcal{C}^e(r') = \ominus$ and p-notblocked(r) and p-grounded(r) then $\mathcal{C}^e(r) = \oplus$*

(C) *if $r \in$ Succ0(r') and $\mathcal{C}^e(r') = \oplus$ and p-notblocked(r) and p-grounded(r) then $\mathcal{C}^e(r) = \oplus$*

(D) *if $r \in$ Succ0(r') and $\mathcal{C}^e(r') = \ominus$ and p-notgrounded(r) then $\mathcal{C}^e(r) = \ominus$.*

We have the following result:

Theorem 2 *Let P be a logic program and let $\mathcal{C}$ and $\mathcal{C}^e : P \mapsto \{\ominus, \oplus\}$ be partial mappings. Then we have if $\mathcal{C}$ is a partial a-coloring of Γ_P and $\mathcal{C}^e$ is an extension of $\mathcal{C}$ as in Definition 7 then $\mathcal{C}_\ominus \subseteq \mathcal{C}^e_\ominus$, $\mathcal{C}_\oplus \subseteq \mathcal{C}^e_\oplus$ and $\mathcal{C}^e$ is a partial a-coloring of Γ_P.*

This theorem gives the conditions for four different propagation cases in arc direction: if a node r is colored c ($c \in \{\ominus, \oplus\}$) then this color can be propagated over 1- and over 0-arcs to the neighbors of r, according to localized propagation conditions (A), (B), (C) and (D).

Now let P be some logic program. Let $\mathcal{C} : P \mapsto \{\ominus, \oplus\}$ be a partial a-coloring. $\mathcal{C}$ is represented by a pair of (disjoint) sets $(\mathcal{C}_\ominus, \mathcal{C}_\oplus)$ s.t. $\mathcal{C}_\ominus = \{r \in P \mid \mathcal{C}(r) = \ominus\}$ and $\mathcal{C}_\oplus = \{r \in P \mid \mathcal{C}(r) = \oplus\}$. Since $\mathcal{C}$ is not total we do not necessarily have $P = \mathcal{C}_\ominus \cup \mathcal{C}_\oplus$. We refer to mapping $\mathcal{C}$ with the pair $(\mathcal{C}_\ominus, \mathcal{C}_\oplus)$ and vice versa. Assume that Γ_P is a global parameter of each of the presented procedure. Let U and N be sets of nodes s.t. U contains the currently uncolored nodes $U = P \setminus (\mathcal{C}_\ominus \cup \mathcal{C}_\oplus)$ and N contains colored nodes whose color has to be propagated. Figures 3, 4 and 5 show the main procedures of `noMoRe` in pseudo code. Notice that procedures **color**$_P$ and **propagate**$_P$ return some extended partial mapping through parameter $\mathcal{C}$ or fail. Procedure **choose**$_P$ either returns some uncolored node r s.t. we have p-grounded(r) wrt the current partial a-coloring or fails. Clearly, **choose**$_P$ implements the non-deterministic part of **color**$_P$. Oppositely, **propagate**$_P$ implements the deterministic consequences of `noMoRe`. Let $L_1(P) = \{r \in P \mid r \in \text{Pred1}(r)\}$ denote the set of all 1-loops in Γ_P. When calling **color**$_P$ the first time, we start with

```
function color_P(U, N: list; C: partial mapping)
 var r : node
   if propagate_P(N, C) then
     U := U \ (C_⊖ ∪ C_⊕)
     if choose_P(U, C, r) then
       U := U \ {r}
       if color_P(U, {r}, (C_⊖, C_⊕ ∪ {r})) then
         return true
       else
         return color_P(U, {r}, (C_⊖ ∪ {r}, C_⊕))
     else
       return propagate_P(U, (C_⊖ ∪ U, C_⊕))
   else
     return false
```

Figure 3: Basic `noMoRe` procedure.

```
function  propagate_P(N:list, C:partial mapping)
 var  r' : node  t : boolean
    t := true
    while ( N ≠ ∅ and t = true ) do
       select r' from N
       if (r' ∈ C_⊕) then
(A)        t := ( propA_P(r', C) and
(C)               propC_P(r', C))
       else
(B)        t := ( propB_P(r', C) and
(D)               propD_P(r', C))
    return t
```

Figure 4: Procedure $\mathbf{propagate}_P$.

$\mathcal{C} = (L_1(P), \mathit{Facts}(P))$, $U = P \setminus (\mathit{Facts}(P) \cup L_1(P))$ and $N = \mathit{Facts}(P) \cup L_1(P)$. That is, we start with all facts colored $\oplus$ and all 1-loops colored $\ominus$. Basically, $\mathbf{color}_P$ takes both a partial mapping $\mathcal{C}$ and a set of uncolored nodes U and aims at coloring these nodes. That is, $\mathbf{color}_P$ computes an extended partial mapping or if this is impossible it fails. This is done by choosing some uncolored node r ($r \in U$) with $\mathbf{choose}_P$ and by trying to color it $\oplus$ first. If this does not give a solution $\mathbf{color}_P$ tries to color node r with $\ominus$. If both possibilities fail $\mathbf{color}_P$ fails. Therefore, we say that node r is used as a *choice*. To be a choice is not a property of a node, because choices are dynamic wrt each solution. Therefore a node may be a choice in one run of $\mathbf{color}_P$ but not in every run, e.g. because there may be heuristics that uses different nodes as choices depending on the actual implementation of $\mathbf{choose}_P$. Observe, that all different a-colorings are obtained via backtracking over choices in $\mathbf{color}_P$.

Notice, that procedure $\mathbf{propagate}_P$ works locally according to conditions (A) to (D) of Definition 7. The color of a node is propagated immediately after getting colored, because the test whether the node was colored correctly is done during propagation. Observe that Theorem 4.1 in [12] implies that partial a-coloring $\mathcal{C}$ can not be extented to some total a-coloring if $\mathbf{propagate}_P$ fails. Therefore, $\mathbf{color}_P$ fails only during propagation. Procedures $\mathbf{propA}_P$, $\mathbf{propB}_P$, $\mathbf{propC}_P$, and $\mathbf{propD}_P$ are the implementations of (A), (B), (C) and (D), respectively. The purpose of $\mathbf{propagate}_P$ is to apply the corresponding propagation cases, e.g. if $\mathcal{C}(r') = \oplus$ then cases (A) and (C) have to be applied[7]. Accoring to Figure 3, partial a-colorings are propagated only in arc direction. That is, if some node is colored then we try to propagate the color of this node to its successors (if possible).

```
function  propB_P(r':node; C:partial mapping)
 var  r : node  S : set of nodes  t : boolean
   S := {r ∈ Succ1(r') | condition (B) holds for r}
   t := true
   while ( S ≠ ∅ and t = true) do
      select r from S
      if r ∈ C_⊖ then
         return false
      else
      if r ∉ C_⊕ then
         t := propagate_P(r, (C_⊖, C_⊕ ∪ {r}))
      else
         return false
   return t
```

Figure 5: Procedure $\mathbf{propB}_P$.

4 Backward Propagation

In [11, 1] it is stated that the number of choices can be reduced by introducing backward propagation, that is, partial a-colorings can also be propagated in opposite arc direction. Clearly, as for propagation in arc direction, we have four backward propagation cases. However, there is a problem with defining localized conditions for backward propagation (as in Definition 7). Assume that Figure 6 depicts a part of some block

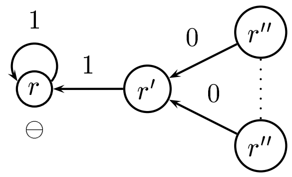

Figure 6: Part of some block graph with partial a-coloring.

graph together with some partial a-coloring. On the one hand, we know that r' has to be colored $\oplus$ (provided that there are no other predecessores of r), becauses this is the only way to block r and if r is not blocked there is no answer set. On the other hand, we cannot color r' with $\oplus$, because we do not have p-grounded(r') (see Definition 6). Therefore we need so-called *transitory a-colorings* .

Definition 8 *Let P be some logic program. We call a partial mapping $\mathcal{C} : P \mapsto \{\ominus, \oplus, +\}$ a* transitory a-coloring *of Γ_P iff $\mathcal{C}$ is an a-coloring of the induced subgraph of Γ_P with nodes $\mathcal{C}_\ominus \cup \mathcal{C}_\oplus$.*

[7] The missing three procedures also used in $\mathbf{propagate}_P$ can be easily implemented analogously to $\mathbf{propB}_P$.

That is, a transitory a-coloring is a partial a-coloring where some nodes may be uncolored or colored with $+$. Color $+$ is used instead of $\oplus$ to color node r' in situations like in Figure 6, where we do not have p-grounded(r') wrt the current partial a-coloring, but r' can still possibly be grounded. In order to transform some transitory a-coloring (during the execution of $\mathbf{color}_P$) to a total a-coloring color $+$ is replaced by color $\oplus$, if possible. This is achieved by propagation. Whenever a node is colored, this color is propagated to all its neighbors immediately, no matter whether these already have been colored or not. In case a node already colored $+$ ($\oplus$) has to be colored $\ominus$ via propagation, propagation fails due to contradiction. When a node already colored $+$ has to be colored $\oplus$ via propagation, color $+$ is simply replaced by $\oplus$. That is, either every color $+$ will become $\oplus$, or $\mathbf{color}_P$ fails. We need the following properties wrt transitory a-colorings:

Definition 9 *Let P be a logic program, let $\mathcal{C}$ be a transitory a-coloring of Γ_P and let $r \in P$ be some node. Then r is* groundable(r) wrt Γ_P and $\mathcal{C}$ *iff $\forall q \in body^+(r)$: $\exists r' \in Pred0(r)$ with $q = head(r')$ s.t. $\mathcal{C}(r') = \oplus$ or r' is uncolored.*

Here groundable(r) means that either r is grounded or that there is some uncolored 0-predecessor, which can possibly be colored $\oplus$ while extending $\mathcal{C}$. For $r \in P$ and $q \in body^+(r)$ we define $S_q \subseteq \mathrm{Pred0}(r)$ as $S_q = \{r' \mid r' \in \mathrm{Pred0}(r)$ and $q = head(r')\}$. Furthermore, for a set of rules $S \subseteq P$ we define p-grounded(S) wrt Γ_P and transitory a-coloring $\mathcal{C}$ iff there is some $r \in S$ s.t. $\mathcal{C}(r) = \oplus$. Now we are ready to define the four localized backward propagation cases.

Definition 10 *Let P be a logic program, let Γ_P be the corresponding block graph and let $\mathcal{C}$ be a transitory a-coloring of Γ_P. We define an extented mapping $\mathcal{C}^e : P \mapsto \{\ominus, \oplus, +\}$ of $\mathcal{C}$ s.t. for each $r \in Dom(\mathcal{C})$ we have $\mathcal{C}^e(r) = \mathcal{C}(r)$ and conditions (A) to (D) of Definition 7 and the following conditions hold for all $r, r' \in P$ wrt Γ_P and $\mathcal{C}^e$:*

(bA) *if $\mathcal{C}^e(r') = \oplus$ and $r \in Pred1(r')$ then $\mathcal{C}^e(r) = \ominus$*

(bB) *if $\mathcal{C}^e(r') = \ominus$ and p-grounded(r') and $r \in Pred1(r')$ s.t. and $\forall r'' \in Pred1(r')$: ($\mathcal{C}^e(r'') = \ominus$ iff $r'' \neq r$) then $\mathcal{C}^e(r) = +$*

(bC) *if $\mathcal{C}^e(r') = \oplus$ and there is some $q \in body^+(r')$ s.t. $q = head(r)$ for some $r \in S_q$ and groundable(r) and for each $r'' \in S_q$: ($\mathcal{C}^e(r'') = \ominus$ iff $r'' \neq r$) then $\mathcal{C}^e(r) = +$*

(bD) *if $\mathcal{C}^e(r') = \ominus$ and p-notblocked(r') and there is some $q \in body^+(r')$ with $q = head(r)$ for some $r \in S_q$ s.t. for each $q' \in body^+(r')$: (p-grounded($S_{q'}$) iff $S_{q'} \neq S_q$) then $\mathcal{C}^e(r) = \ominus$.*

Intuitively, these cases ensure that an already $\oplus$-colored node is grounded (bC) and not blocked (bA) while an already $\ominus$-colored node is blocked (bB) or not grounded (bD). So in a sense, the purpose of these cases is to justify the color of a node. Observe, that cases (bB) and (bC) use color $+$ instead of $\oplus$ (see Defintion 8). We have the following result corresponding to Theorem 2:

Theorem 3 *Let P be a logic program and let $\mathcal{C}$ and $\mathcal{C}^e : P \mapsto \{\ominus, \oplus, +\}$ be a partial mappings. Then we have if $\mathcal{C}$ is a transitory a-coloring of Γ_P and $\mathcal{C}^e$ is an extension of $\mathcal{C}$ as in Definition 10 then $\mathcal{C}_\ominus \subseteq \mathcal{C}^e_\ominus$, $\mathcal{C}_\oplus \subseteq \mathcal{C}^e_\oplus$ and $\mathcal{C}^e$ is a transitory a-coloring of Γ_P.*

Let us show how $\mathbf{color}_P$ computes the a-colorings of the block graph of program (1) (see Figure 1). At the beginning we cannot propagate anything, because there is no fact and no 1-loop. We take r_a as a choice. First, we try to color r_a with $\oplus$ by calling $\mathbf{color}_P(U, N, \mathcal{C})$ with $U = P \setminus \{r_a\}$, $N = \{r_a\}$ and $\mathcal{C} = (\emptyset, \{r_a\})$. Now, $\mathbf{propagate}_P(N, \mathcal{C})$ is executed. By propagating $\mathcal{C}(r_a) = \oplus$ with case (A) we get $\mathcal{C}(r_b) = \ominus$ and $\mathcal{C}(r_d) = \ominus$. Recursively, through case (B) $\mathcal{C}(r_c) = \oplus$ is propagated. This gives $\mathcal{C} = (\{r_b, r_d\}, \{r_a, r_c\})$. Since U becomes the empty set, $\mathbf{choose}_P$ fails and $\mathcal{C}$ is the first output. So far we did not need backward propagation.

Now, we color r_a with $\ominus$ through calling $\mathbf{color}_P(U, N, \mathcal{C})$ with U and N as above and $\mathcal{C} = (\emptyset, \{r_a\})$. Since no (backward) propagation is possible we have to compute the next choice. For $\mathbf{choose}_P$ all three uncolored nodes are possible choices s.t. **CP** holds. Assume $\mathcal{C}(r_b) = \oplus$ as next choice. Through propagation case (A) we get $\mathcal{C}(r_c) = \ominus$. This color of r_c has to be propagated by executing $\mathbf{propagate}_P(\{r_c\}, (\{r_a, r_c\}, \{r_c\}))$. By using propagation case (B) we obtain $\mathcal{C}(r_d) = \oplus$. Recursively, propagation of $\oplus$ for r_d gives no contradiction and $\mathcal{C} = (\{r_a, r_c\}, \{r_b, r_d\})$ is the second a-coloring. By assuming $\mathcal{C}(r_b) = \ominus$, that is, $\mathcal{C} = (\{r_a, r_b\}, \emptyset)$, r_c is colored with $\oplus$ through backward propagation case (bB). By propagation of this color with case (A) node r_d is colored with $\ominus$. By applying case (B) to the color of r_d we obtain that r_a has to be colored with $\oplus$, because it is not blocked, but this is a contradiction to $\mathcal{C}(p_a) = \ominus$. Thus, there is no further solution and we have found the two solutions with two choices. Observe that the usage of (bB) saves one additional choice, since without backward propagation the partial coloring $\mathcal{C} = (\{r_a, r_b\}, \emptyset)$ could not have been extended any more and another choice would have been necessary.

5 Jumping

```
procedure propagate_P(N:list, C:part. mapping)
  var r' : node;
    while N ≠ ∅ do
      select r' from N;
      if (r' ∈ C_⊕) then
(A)       if propA_P(r', C) fails then fail;
(C)       if propC_P(r', C) fails then fail;
(bA)      if backpropA_P(r', C) fails then fail;
(bC)      if backpropC_P(r', C) fails then fail;
          if jumpC_P(r', C) fails then fail;
       else
(B)       if propB_P(r', C) fails then fail;
(D)       if propD_P(r', C) fails then fail;
(bB)      if backpropB_P(r', C) fails then fail;
(bD)      if backpropD_P(r', C) fails then fail;
          if jumpB_P(r', C) fails then fail;
          if jumpD_P(r', C) fails then fail;

procedure jumpB_P(r':node; C:partial mapping,)
  var S : set of nodes;
    S := Succ1(r')
    while S ≠ ∅ do
      select r' from S;
      if C(r') = ⊖ then backpropB_P(r', C);
```

Figure 7: Extended propagation procedures including backward propagation and jumping.

A further improvement for the rule-based algorithm is so-called jumping. Backward propagation according to (bB), (bC) and (bD) requires certain conditions to be fulfilled, which may not be known when a node is colored. For (bA) this is not the case, because in (bA) there is no further condition. Take the following program P and its corresponding block graph Γ_P (see Figure 8):

$$P = \left\{ \begin{array}{ll} \multicolumn{2}{c}{a \leftarrow not\ a, not\ b, not\ d} \\ b \leftarrow not\ c & d \leftarrow not\ e \\ c \leftarrow not\ b & e \leftarrow not\ d \end{array} \right\} \tag{2}$$

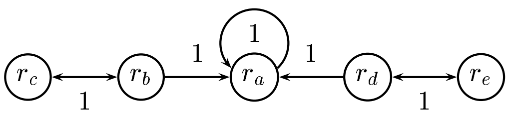

Figure 8: Block graph of program (2).

We know that $\mathcal{C}(\{r_a\}) = \ominus$, otherwise there would not be an answer set at all. Since r_a is trivially grounded, it has to be blocked. This can be achieved by two rules, r_b and r_d, though one is sufficient. But we do not know yet, which one should be colored $\oplus$. Later on, when e.g. r_b has been used as a choice and is colored $\ominus$ this is achieved via jumping, that is, case (bB) is used again for node r_a and r_d is colored $\oplus$. Finally, with (A) node r_e is colored $\ominus$. In this way jumping helps to avoid unnecessary choices, because without jumping we would need another choice to color nodes r_d and r_e. Backward propagation would not be complete without jumping. In general, whenever a $\ominus$ is propagated along a 1-arc to an already $\ominus$-colored node, we check backward propagation case (bB) for this node again. Similarly, we check (bC) and (bD) again for $\oplus$-colored and $\ominus$-colored 0-successors of already colored and propagated nodes, respectively.

As an example, Figure 7 shows the implementation of procedure **jumpB**$_P$, which jumps to an already colored node in order to check backward propagation (bB) again. By replacing procedure **propagate**$_P$ in Figure 3 with **propagate**$_P$ in Figure 7 we obtain an algorithm for computing a-colorings including backward propagation and jumping. The four procedures for the backward propagation cases are implemented similarly to procedure **propB**$_P$ in Figure3, with the exception that we aim at coloring predecessors instead of successors now.

6 Results

For a transitory mapping $\mathcal{C} : P \mapsto \{\ominus, \oplus, +\}$ we define the set of *corresponding answer sets* $A_\mathcal{C}$ as

$$\begin{array}{rcl} A_\mathcal{C} & = & \{X \mid X \text{ is answer set of } P \\ & & \quad \text{and } \mathcal{C}_\oplus \subseteq GR(P,X) \text{ and } \mathcal{C}_\ominus \cap GR(P,X) = \emptyset\}. \end{array}$$

If $\mathcal{C}$ is undefined for all nodes then $A_\mathcal{C}$ contains all answer sets of P. If $\mathcal{C}$ is a total mapping s.t. no node is colored with $+$ then $A_\mathcal{C}$ contains exactly one answer set of P (if $\mathcal{C}$ is an a-coloring). With this notation we formulate the following result:

Theorem 4 *Let P be a logic program, let $\mathcal{C}$ and $\mathcal{C}'$: $P \mapsto \{\ominus, \oplus, +\}$ be transitory mappings. Then for each $r \in (\mathcal{C}_\ominus \cup \mathcal{C}_\oplus \cup \mathcal{C}_+)$ we have if* **propagate**$_P(\{r\}, \mathcal{C})$ *succeeds and $\mathcal{C}'$ is the transitory mapping after its execution then $A_\mathcal{C} = A_{\mathcal{C}'}$.*

This theorem states that **propagate**$_P$ neither discards nor introduces answer sets corresponding to some transitory mapping $\mathcal{C}$. Hence, only nodes used as choices lead to different answer sets.

Finally, let C_P be the set of all solutions of **color**$_P$. We obtain correctness and completeness of **color**$_P$.

Theorem 5 *Let P be a logic program, let Γ_P be its block graph, let $\mathcal{C} : P \mapsto \{\ominus, \oplus\}$ be a mapping and let C_P the set of all solutions of* $\mathbf{color}_P$ *for program P. Then $\mathcal{C}$ is an a-coloring of Γ_P iff $\mathcal{C} \in C_P$.*

7 Experiments

As benchmarks, we have used some instances of NP-complete problems proposed in [2], namely, the independent set problem for circle graphs[8], the problem of finding Hamiltonian cycles in complete graphs and the problem of finding classical graph colorings. Furthermore we have tested some planning problems taken from [6] and the n-queens problem. In Table 1 we have counted the number of choices instead of measuring time, since the number of choices indicates how good an algorithm deals with a non-deterministic problem. For `smodels` results with and without lookahead (results in parenthesis) are shown[9].

	`noMoRe`			`smodels`	
backprop	no	yes	yes		
jumping	no	no	yes		
ham_k_7	14335	14335	2945	4814	(34077)
ham_k_8	82200	82200	24240	688595	(86364)
ind_cir_20	539	317	276	276	(276)
ind_cir_30	9266	5264	4609	4609	(4609)
p1_step4	-	464	176	7	(69)
p2_step6	-	13654	3779	75	(3700)
col4x4	27680	27680	7811	7811	(102226)
col5x5	-	-	580985	580985	(2.3 Mil)
queens4	84	84	5	1	(11)
queens5	326	326	13	9	(34)

Table 1: Number of choices (all solutions) for different problems.

The influence of backward propagation and jumping on the number of choices is clearly visible. There are also some problems where we did not obtain a solution after more than 12 hours without backward propagation. Table 1 impressively shows that `noMoRe` with backward propagation and jumping is now comparable with `smodels` on several problem classes; especially if we disable the lookahead of `smodels`. The difference between `smodels` and `noMoRe` for planning examples and the n-queens problems seems to come from different heuristics for making choices. We have just started to investigate the influence of more elaborated heuristics for making choices.

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

[9]Since we feel that `dlv` does not give sufficient information on choices there are no results for `dlv` in Table 1. Whereas `smodels` and `noMoRe` make exactely one choice at each (choice) point in the search space, `dlv` makes several choices at the same point of the search space.

8 Conclusion

We have shown that by introducing backward propagation together with jumping the rule-based algorithm implemented in `noMoRe` can be greatly improved. A related method of backward propagation wrt answer set semantics for normal logic programs was proposed in [9]. However, a lot of the obtained improvement is due to the concept of a third color $+$. There seems to be a close relation between `noMoRe`'s color $+$ and `dlv`'s must-be-true truth value [6], though this has to be studied more thoroughly, because `noMoRe` is rule-based and `dlv` (and `smodels`) is literal-based. Through the conducted experiments the impact of the improvements is shown. `NoMoRe` is now compareable to `smodels` on many different problem classes measuring the number of choices. This improvement was obtained by improving the deterministic consequences of `noMoRe`. However, there are still some interesting open questions. The main one is whether rule-based computation of answer sets is different from atom-based (literal-based) or not. During our experiments we have detected programs (with a large rule atom ratio) for which atom-based computations are more suitable and other programs (with a small rule atom ratio) for which rule-based computation performs better. Currently, we have no general answer to this question and a general comparison between atom-based and rules-based methods for logic programs will be necessary.

Another question for future work is whether rule-based approaches like implemented in `noMoRe` can lead to a system which is able to compete (considering time) against state of the art systems like `smodels` and `dlv`. Clearly, this question cannot be answered without having a system for the rule-based approach that is as elaborated implemented as `smodels` and `dlv` are. Right now for `noMoRe` this is not the case, because its development is in a state where `smodels` and `dlv` were some years ago, for example the heuristics of `noMoRe` still does not have a lookahead. In fact, the only very simple heuristic we use so far in `noMoRe` is to make choices for the color of a node r only if we have p-grounded(r). However, we think that our experiments show that there is a chance for rule-based methods to be able to compete against atom-based in the future. Therefore further work will also include the study of heuristics and different implementation techniques.

9 Acknowledgements

We would like to thank Ph. Besnard, K. Wang and T. Schaub 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

[1] C. Anger, K. Konczak, and T. Linke. `NoMoRe`: A system for non-monotonic reasoning with logic programs under answer set semantics. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Automated Reasoning*, volume 2083 of *Lecture Notes in Computer Science*, pages 325–330. Springer, 2001. First International Joint Conference on Automated Reasoning.

[2] P. Cholewiński, V. Marek, A. Mikitiuk, and M. Truszczyński. Experimenting with nonmonotonic reasoning. In *Proceedings of the International Conference on Logic Programming*, pages 267–281. MIT Press, 1995.

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

[4] D. East and M. Truszczyński. dcs: An implementation of datalog with constraints. In *Proceedings of the National Conference on Artificial Intelligence.* MIT Press, 2000.

[5] 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, *Proceedings of the Fourth International Conference on Logic Programming and Non-Monotonic Reasoning*, volume 1265 of *Lecture Notes in Artificial Intelligence*, pages 363–374. Springer Verlag, 1997.

[6] W. Faber, N. Leone, and G. Pfeifer. Pushing goal derivation in dlp computations. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, volume 1730 of *Lecture Notes in Artificial Intelligence*, pages 177–191, El Paso, Texas, USA, 1999. Springer Verlag.

[7] W. Faber, N. Leone, and G. Pfeifer. Experimenting with heuristics for answer set programming. In B. Nebel, editor, *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 635–640. Morgan Kaufmann Publishers, 2001.

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

[9] N. Iwayama and K. Satoh. Computing abduction by using tms with top-down expectation. *Journal of Logic Programming*, 44:179–206, 2000.

[10] V. Lifschitz. Answer set planning. In *Proceedings of the 1999 International Conference on Logic Programming*, pages 23–37. MIT Press, 1999.

[11] T. Linke. Graph theoretical characterization and computation of answer sets. In B. Nebel, editor, *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 641–645. Morgan Kaufmann Publishers, 2001.

[12] T. Linke. Rule-based computation of answer sets. 2002. submitted.

[13] 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, pages 5–19, Lisbon, Portugal, 1998. Springer Verlag.

[14] 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.

[15] 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. of the Fourth International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 420–429. Springer, 1997.