Pruning Operators for Answer Set Programming Systems

Francesco Calimeri
University of Calabria
87030 Rende (CS), Italy
calimeri@mat.unical.it

Wolfgang Faber
TU Wien
A-1040 Wien, Austria
faber@kr.tuwien.ac.at

Nicola Leone
University of Calabria
87030 Rende (CS), Italy
leone@unical.it

Gerald Pfeifer
TU Wien
A-1040 Wien, Austria
pfeifer@dbai.tuwien.ac.at

Abstract

Answer Set Programming (ASP) is a novel paradigm in Logic Programming, which allows for solving problems in a simple and highly declarative way. The language of ASP (function-free disjunctive logic programming) is very expressive and supports the representation of problems of high computational complexity (specifically, all problems in the complexity class $\Sigma_2^P = \mathrm{NP}^{\mathrm{NP}}$). Importantly, the ASP encoding of a large variety of problems is often very concise, simple, and elegant.

In this paper, we explain the computational process performed by ASP systems, with a focus on search space pruning, which is crucial for efficiency. We analyze the properties of two main pruning operators, namely (*Fitting's operator* and *Well-founded operator*), discuss their peculiarities and differences with respect to efficiency and effectiveness. We design an intelligent strategy for combining the two operators, which exploits the advantages of both. We implement our approach in the ASP system **DLV**, and perform some experiments. The experiments show interesting results, and evidence how the choice of the pruning operator affects the performance of ASP systems.

1 INTRODUCTION

The high expressiveness of answer set programming (which allows for representing all problems in the complexity class $\Sigma_2^P = \mathrm{NP}^{\mathrm{NP}}$ [Eiter et al., 1997]) comes at the price of a high computational cost in the worst case, which makes the implementation of efficient ASP systems a difficult task. Still some efforts have been made in this direction. After some pioneering work on stable model computation [Bell et al., 1994, Subrahmanian et al., 1995], a number of modern ASP systems are now available. The two most widespread ASP systems are **DLV** [Faber et al., 1999a, Faber et al., 2001] and Smodels [Niemelä, 1999, Simons, 2000]; but also other systems support ASP to some extent, including `NoMoRe` [Anger et al., 2001], QUIP [Egly et al., 2000], CCALC [McCain and Turner, 1998], XSB [Rao et al., 1997], and DCS [East and Truszczyński, 2000]. Nevertheless, much work has to be done to make ASP systems fully satisfactory for modern knowledge-based applications. The design of new optimization techniques and smart algorithms for the computation of ASP programs is of utmost importance. The present paper goes in this direction, focusing on search space pruning, an extremely critical problem for the efficiency of ASP systems. The main contributions of the paper are as follows:

- We describe the main steps of the computational process performed by ASP systems with a focus on search space pruning, which is crucial for efficiency. We analyze the properties of the disjunctive extensions of two well-known pruning operators for ASP, *Fitting*'s operator and the *Well-founded* operator. We discuss their strengths and weaknesses w.r.t. efficiency and effectiveness.

- We design an intelligent strategy for combining these two pruning operators, which exploits the advantages of both, starting from several known results established in different works on ASP and focusing on: modularity properties [Lifschitz and Turner, 1994, Eiter et al., 1997, Leone et al., 1997], head-cycle free programs [Ben-Eliyahu and Dechter, 1994], acyclic programs [Fages, 1994], disjunctive unfounded sets and complexity [Leone et al., 1997], combining them in a smart way.

- We implement our approach in the ASP system **DLV**, taking care of efficiency issues and respecting the known complexity bounds. Indeed, the fixpoints of Fitting's operator are computed in linear time, as are the Greatest Unfounded Sets (which contribute the negative inferences in the Well-founded operator).
- We report experimental results on a number of benchmark problems to assess the validity of our approach. The experiments show that the choice of the pruning operator has a strong impact on the performance of ASP systems, and specifically that our techniques considerably improve the efficiency of the ASP system **DLV**.

To our knowledge, this is the first paper that focuses on pruning the search space for *disjunctive* ASP programs. A number of related works have studied pruning operators in the domain of non-disjunctive ASP programs. The use of the well-founded operator in the computation of non-disjunctive programs has been first proposed in [Leone et al., 1993] while its concrete implementation in a system was first done in [Subrahmanian et al., 1995]. The well-founded operator corresponds to the *upper closure* operator of the Smodels system [Niemelä, 1999], which has been implemented very efficiently in Smodels, employing a novel optimization technique to localize the computation [Simons, 2000].

2 ASP PROGRAMS AND NOTATION

A *(disjunctive) rule* r is a formula

$$a_1 \ \vee \ \cdots \ \vee \ a_n \ \text{:-}\ b_1, \cdots, b_k,\ \mathtt{not}\ b_{k+1}, \cdots,\ \mathtt{not}\ b_m.$$

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are classical literals (atoms possibly preceded by the classical negation symbol $\neg$), and $n \geq 0$, $m \geq k \geq 0$. Given a rule r, let $H(r) = \{a_1, ..., a_n\}$ denote the set of head literals, $B^+(r) = \{b_1, ..., b_k\}$ and $B^-(r) = \{\mathtt{not}\ b_{k+1}, ..., \mathtt{not}\ b_m\}$ the set of positive and negative body literals, resp., and $B(r) = B^+(r) \cup B^-(r)$.

A rule r with $B^-(r) = \emptyset$ is called *positive*; a rule with $H(r) = \emptyset$ is referred to as *integrity constraint*. If the body is empty we usually omit the :- sign. Comparison operators (like $=, <, >, <>$) are built-in predicates and may appear in rule bodies.

An *(ASP) program* $\mathcal{P}$ is a finite set of rules; $\mathcal{P}$ is a *positive* program if all rules in $\mathcal{P}$ are positive (i.e., not-free). An object (atom, rule, etc.) containing no variables is called *ground* or *propositional*.

Given a literal l, let $\mathtt{not}.l = a$ if $l = \mathtt{not}\ a$, otherwise $\mathtt{not}.l = \mathtt{not}\ l$, and given a set L of literals, $\mathtt{not}.L = \{\mathtt{not}.l \mid l \in L\}$.

Herbrand Universe, *Herbrand Base*, and the *ground instantiation Ground*($\mathcal{P}$) of $\mathcal{P}$ are defined as usual. The semantics of an ASP program is given by its (consistent) answer sets - an extension of the concept of stable models to the realm of ASP programs proposed by Gelfond and Lifschitz in [Gelfond and Lifschitz, 1991], for a nice introduction we refer to [Lifschitz, 1996].

Example 2.1 *For the program* $\mathcal{P} = \{a \vee \neg\ b \vee c.,\ \text{:-}\ a.,\ \neg\ b \ \text{:-}\ c.,\ c\ \text{:-}\ \neg\ b.\ \}$, $I_1 = \{a, \neg\ b\}$ *and* $I_2 = \{c\}$ *are interpretations, but* $M = \{\neg\ b, c\}$ *is the only answer set.*

3 ANSWER SETS COMPUTATION

In this section, we describe the main steps of the computational process performed by ASP systems. We will describe the computational engine of the **DLV** system [Faber et al., 1999a, Faber et al., 2001] which will be used for the experiments, but also other ASP systems like Smodels [Niemelä and Simons, 1996, Simons, 2000] employ a very similar procedure.

In general, an answer set program $\mathcal{P}$ contains variables. The computational step of an ASP system eliminates these variables, generating a ground instantiation *ground*($\mathcal{P}$) of $\mathcal{P}$ which is a (usually much smaller) subset of all syntactically constructible instances of the rules of $\mathcal{P}$ having precisely the same answer sets as $\mathcal{P}$ [Faber et al., 1999a]. In this phase, also true negation is eliminated by a straightforward rewriting pass (so in the following "atoms" might be rewritten classical literals). The hard part of the computation is then performed on this simplified ground program by the Model Generator, which is sketched in Figure 1. For brevity, $\mathcal{P}$ refers to the simplified ground program in the sequel.

Roughly, the Model Generator produces some "candidate" answer sets. Each candidate I is then verified by the function *IsAnswerSet(I)*, which checks whether I is a minimal model of the program $\mathcal{P}^I$ obtained by applying the GL-transformation w.r.t. I.

The interpretations handled by the Model Generator are partial interpretations. A *partial* interpretation I is a consistent set of atoms, possibly preceded by the default negation symbol (not). A ground literal L is true w.r.t. I if $L \in I$; it is false w.r.t. I if $\mathtt{not}.L \in I$; L

```
Function ModelGenerator(var I: Interpretation): bool;
var   inconsistency: bool;
begin
      DetCons(I,inconsistency);
      if inconsistency then return false;
      if"no atom is undefined in I" then
        return IsAnswerSet(I);
      Select an undefined atom A using a heuristic;
      if ModelGenerator(I ∪ {A}) then return true;
      else return ModelGenerator(I ∪ {not A});
end;
```

Figure 1: Computation of Answer Sets

is undefined otherwise. Initially, the ModelGenerator function is invoked with I set to the empty interpretation (all atoms are undefined at this stage). If the program $\mathcal{P}$ has an answer set, then the function returns true and sets I to the computed answer set; otherwise it returns false. The Model Generator is similar to the Davis-Putnam procedure in SAT solvers. It first calls a function DetCons, which extends I with those literals that can be deterministically inferred. This is similar to unit propagation as employed by SAT solvers, but exploits the peculiarities of ASP for making further inferences (e.g., it uses the knowledge that every answer set is a minimal model).

If DetCons does not detect any inconsistency, an atom A is selected according to a heuristic criterion and ModelGenerator is recursively called on both $I \cup \{A\}$ and $I \cup \{\texttt{not}\ A\}$. The atom A corresponds to a branching variable of a SAT solver.

For the performance of an ASP system the implementation of DetCons is crucial in two ways: It has to perform its task as quickly as possible, while pruning the search space as much as possible.

4 PRUNING OPERATORS

In this section we review two operators that are useful to implement DetCons. As already mentioned, DetCons has to expand a given interpretation as much as possible to reduce the search space, while ensuring that such an expansion never causes any answer set to be missed. In other words, if an interpretation I is contained in an answer set M, that answer set will also contain the expansion of I computed by DetCons. We can state this "safety" property formally for a generic operator $\Gamma_{\mathcal{P}}$: for each interpretation I and each answer set M of a given program $\mathcal{P}$, $I \subseteq M$ iff $\Gamma_{\mathcal{P}}(I) \subseteq M$.

The two operators in question are the *Fitting* ($\Phi_{\mathcal{P}}$) operator and the *Well-founded* ($\mathcal{W}_{\mathcal{P}}$) operator. Both have the property described above, and extend the two corresponding operators defined for disjunction-free programs [Fitting, 1985, van Gelder et al., 1991] to the class of disjunctive logic programs. Both $\Phi_{\mathcal{P}}$ and $\mathcal{W}_{\mathcal{P}}$ consist of two parts: The part drawing positive inferences (which is an extension of the immediate consequence operator $T_{\mathcal{P}}$, defined for three-valued interpretations of normal logic programs [van Gelder et al., 1991], to disjunctive programs) is the same for both operators; they only differ in the way they perform negative inferences.

Definition 4.1 Let $\mathcal{P}$ be a program, and I a (partial) interpretation.

$$\begin{aligned}\mathcal{T}_{\mathcal{P}}(I) = \{a \mid \exists r \in \mathcal{P}\ s.t.\ a \in H(r) :\\ H(r) - \{a\} \subseteq \texttt{not}.I \wedge B(r) \subseteq I\}\end{aligned}$$

Example 4.1 *Consider the program* $\mathcal{P}_1 = \{a \vee b.,\ c \text{ :- } \texttt{not}\ a.\ ,\ d \text{ :- } e.,\ \ e \text{ :- } d.,\ \ k \text{ :- } \texttt{not}\ e.\}$. *Suppose* $I = \{\texttt{not}\ a\}$, *then* $\mathcal{T}_{\mathcal{P}}(I) = \{b, c\}$.

4.1 FITTING ($\Phi_{\mathcal{P}}$) OPERATOR

$\Phi_{\mathcal{P}}$ extends the *Fitting* operator [Fitting, 1985] to the disjunctive case. Intuitively, $\Phi_{\mathcal{P}}$ derives falsity of those atoms for which there is no rule left that could be used to derive them as true.

Definition 4.2 Let $\mathcal{P}$ be a program, and I a (partial) interpretation. Define $\gamma_{\mathcal{P}}$, $\Phi_{\mathcal{P}}$ and $\{F_n\}$ as follows.

$$\begin{aligned}\gamma_{\mathcal{P}}(I) = {} & \{a \mid \forall r \in \mathcal{P}\ s.t.\ a \in H(r) :\\ & (H(r) - \{a\}) \subseteq I \vee B(r) \cap \texttt{not}.I \neq \emptyset\}\\ \Phi_{\mathcal{P}}(I) = {} & \mathcal{T}_{\mathcal{P}}(I) \cup \texttt{not}.\gamma_{\mathcal{P}}(I)\\ F_0 = I, \quad & F_k = F_{k-1} \cup \Phi_{\mathcal{P}}(F_{k-1}) \quad \textit{for}\ k > 0\end{aligned}$$

$F_0, \ldots, F_n$ is a growing sequence whose *n-th* term is the *n-fold* application of $\Phi_{\mathcal{P}}$ to I. The least fixpoint $\Phi^{\infty}_{\mathcal{P}}(I)$ of $\Phi_{\mathcal{P}}$ containing I, is defined as the limit to which $\{F_n\}_{n \in \mathcal{N}}$ converges.

Example 4.2 *Consider the program* $\mathcal{P}_1$ *of Example 4.1, and the interpretation* $I = \{a\}$. *Here* $\gamma_{\mathcal{P}}(I) = \{b, c\}$. *It is easy to see that* $\Phi_{\mathcal{P}}(I) = \emptyset \cup \texttt{not}.\{b, c\} = \{\texttt{not}\ b, \texttt{not}\ c\}$. *We thus obtain* $F_0 = \{a\}, F_1 = \{a, \texttt{not}\ b, \texttt{not}\ c\}, F_2 = F_1 = \Phi^{\infty}_{\mathcal{P}}(I)$.

Proposition 4.1 *For every answer set* M *of a given program* $\mathcal{P}$, *an interpretation* $I \subseteq M$ *iff* $\Phi^{\infty}_{\mathcal{P}}(I) \subseteq M$.

Importantly, for any interpretation I, $\Phi^{\infty}_{\mathcal{P}}(I)$ is *linear-time computable.* The $\Phi_{\mathcal{P}}$ operator seems to be a good choice as a pruning operator, as it is "*safe*", has the capability to perform negative inferences, and its fixpoint $\Phi^{\infty}_{\mathcal{P}}(I)$ is efficiently computable.

Unfortunately, $\Phi_{\mathcal{P}}$ fails to derive all possible negative consequences. For instance, in Example 4.2 it fails to derive d and e as false w.r.t. I while the only rules having these atoms in the head will never have a true body. The *Well-founded* operator presented in the following section is "stronger" in this respect.

4.2 WELL-FOUNDED ($\mathcal{W}_{\mathcal{P}}$) OPERATOR

The $\mathcal{W}_{\mathcal{P}}$ operator defined in [Leone et al., 1997] extends the operator defined in [van Gelder et al., 1991] (whose least fixpoint is the well-founded model) to the disjunctive case. It is defined by an extension of the notion of *unfounded sets* to disjunctive logic programs.

Definition 4.3 Let I be a (partial) interpretation for a program $\mathcal{P}$. A set $X \subseteq B_{\mathcal{P}}$ of ground atoms is an *unfounded set* for $\mathcal{P}$ w.r.t. I if, for each $a \in X$ and for each rule $r \in \mathcal{P}$ such that $a \in H(r)$, at least one of the following conditions holds: (i) $B(r) \cap \texttt{not}.I \neq \emptyset$, (ii) $B^+(r) \cap X \neq \emptyset$, (iii) $(H(r) - X) \cap I \neq \emptyset$.

While for non-disjunctive programs the union of unfounded sets is again an unfounded set for all interpretations, this does not hold in general for disjunctive programs (see [Leone et al., 1997]). For instance, for $\mathcal{P} = \{a \vee b\}$ and $I = \{a, b\}$, both $\{a\}$ and $\{b\}$ are unfounded sets w.r.t. I; but their union $\{a, b\}$ is not. Let $\mathbf{I}_{\mathcal{P}}$ denote the set of all interpretations of $\mathcal{P}$ for which the union of all unfounded sets for $\mathcal{P}$ w.r.t. I is an unfounded set for $\mathcal{P}$ w.r.t. I as well. Given $I \in \mathbf{I}_{\mathcal{P}}$, let $GUS_{\mathcal{P}}(I)$ (the *greatest unfounded set* of $\mathcal{P}$ w.r.t. I) denote the union of all unfounded sets for $\mathcal{P}$ w.r.t. I. We can now introduce the *Well-founded* operator.

Definition 4.4 Let $\mathcal{P}$ be a program, and $I \in \mathbf{I}_{\mathcal{P}}$ a (partial) interpretation. Define $\mathcal{W}_{\mathcal{P}}$ and $\{W_n\}$ as follows:

$$\mathcal{W}_{\mathcal{P}}(I) = \mathcal{T}_{\mathcal{P}}(I) \cup \texttt{not}.GUS_{\mathcal{P}}(I).$$
$$W_0 = I, \quad W_k = W_{k-1} \cup \mathcal{W}_{\mathcal{P}}(W_{k-1}) \quad \textit{for } k > 0$$

$W_0, \ldots, W_n$ is a growing sequence whose *n-th* term is the *n-fold* application of $\mathcal{W}_{\mathcal{P}}$ to I. The least fixpoint $\mathcal{W}^{\infty}_{\mathcal{P}}(I)$ of $\mathcal{W}_{\mathcal{P}}$ containing I is the limit to which $\{W_n\}_{n \in \mathcal{N}}$ converges. Note that $\mathcal{W}^{\infty}_{\mathcal{P}}(I) \in \mathbf{I}_{\mathcal{P}}$ if $I \in \mathbf{I}_{\mathcal{P}}$.

Example 4.3 *Considering the program $\mathcal{P}_1$ of Example 4.1 and the interpretation $I = \{a\}$, we get $GUS_{\mathcal{P}}(I) = \{b, c, d, e\}$. b is added because of (iii), and c because of (i) in Definition 4.3. Then d and e appear in the head of only a single rule each and for both (ii) of Definition 4.3 holds. We obtain $\mathcal{W}_{\mathcal{P}}(I) = \{\texttt{not } b, \texttt{not } c, \texttt{not } d, \texttt{not } e\}$ and $W_0 = \{a\}, W_1 = \{a, \texttt{not } b, \texttt{not } c, \texttt{not } d, \texttt{not } e\}, W_2 = \{a, \texttt{not } b, \texttt{not } c, \texttt{not } d, \texttt{not } e, k\}, W_3 = W_2 = \mathcal{W}^{\infty}_{\mathcal{P}}(I)$.*

Proposition 4.2 [Leone et al., 1997] Let I be an interpretation for a program $\mathcal{P}$, and let M be an answer set for $\mathcal{P}$. If $I \subseteq M$, then (a) I belongs to the domain $\mathbf{I}_{\mathcal{P}}$ of $\mathcal{W}_{\mathcal{P}}$, and (b) $\mathcal{W}_{\mathcal{P}}(I) \subseteq M$.

The drawback of $\mathcal{W}_{\mathcal{P}}$ in the disjunctive case is that it is applicable only for interpretations in the class $\mathbf{I}_{\mathcal{P}}$. Deciding whether $I \in \mathbf{I}_{\mathcal{P}}$ is co-NP-hard [Scarcello, 1997, Leone et al., 1997], and cannot be efficiently tested unless P = NP.

5 PRUNING OPERATORS ON SYNTACTICALLY RESTRICTED CLASSES OF PROGRAMS

In this section, we explore several interesting properties of the operators on some restricted (syntactic) classes of programs. To this end, we introduce *dependency graphs* which are based on the dependencies of head predicates on the positive body predicates of a rule.

With every program $\mathcal{P}$, we associate a directed graph $DG_{\mathcal{P}} = (\mathcal{N}, E)$, called the *dependency graph* of $\mathcal{P}$, where (i) each predicate of $\mathcal{P}$ is a node in $\mathcal{N}$, and (ii) there is an arc in E directed from node a to node b if there is a rule r in $\mathcal{P}$ such that two predicates a and b of literals appear in $B^+(r)$ and $H(r)$, respectively.

A *component* is a maximal strongly connected subset of nodes. Since there is a one-to-one correspondence between nodes in $DG_{\mathcal{P}}$ and predicates in $\mathcal{P}$, each component of $DG_{\mathcal{P}}$ is related to a subset of predicates (which we call a "component" of $\mathcal{P}$ as well). Those rules in $\mathcal{P}$ which have a head predicate in C, form a *subprogram* of $\mathcal{P}$ restricted to C[1]. We denote this subprogram by $\mathcal{P}_C$. By $Comp(\mathcal{P})$ we denote the set of all "components", again referring to both predicates and related subprograms.

Given a program $\mathcal{P}$ and its dependency graph $DG_{\mathcal{P}}$, we say that:

- a component C is *cyclic* if the related subprogram $\mathcal{P}_C$ of $\mathcal{P}$ contains at least one recursive rule (i.e., a rule r such that a head predicate and a positive body predicate of r are in C); C is *acyclic* if it is not cyclic.
- a component C is *head-cycle-free* (HCF) iff the related subprogram $\mathcal{P}_C$ of $\mathcal{P}$ contains no rule r such that two predicates occurring in the head of r belong to C.

[1] Note that a disjunctive rule may occur in the subprograms of two different components.

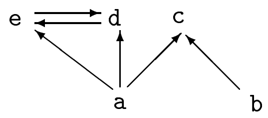


Figure 2: *$DG_{\mathcal{P}_1}$ graph*

$DG_{\mathcal{P}}$ and $\mathcal{P}$ are *cyclic* if there is *at least one cyclic* component, otherwise they are acyclic. They are HCF if *all* components are HCF.

Example 5.1 *Consider the following program $\mathcal{P}_1$:*

$$\{ \ a \vee b., \quad c \text{ :- } a. \ , \quad c \text{ :- } b. \ , \quad d \vee e \text{ :- } a. \ , \\ d \text{ :- } e., \quad e \text{ :- } d, \texttt{not } b. \ \}$$

The dependency graph $DG_{\mathcal{P}_1}$ of $\mathcal{P}_1$ is depicted in Figure 2. There are four components: $\{a\}$, $\{b\}$, $\{c\}$, $\{d, e\}$. All of them are acyclic except for the last which is also the only non-HCF component, as the head of $d \vee e$:- a. contains two predicates belonging to the same cycle. The whole graph, and thus the program, is cyclic but not HCF.

We next present two well-known theorems about the operators $\Phi^{\infty}_{\mathcal{P}}(I)$ and $\mathcal{W}^{\infty}_{\mathcal{P}}(I)$, the importance of which will become clear in the next section.

Theorem 5.1 Given a program $\mathcal{P}$ and an interpretation I, if $\mathcal{P}$ is *acyclic* then $\mathcal{W}_{\mathcal{P}}(I) = \Phi_{\mathcal{P}}(I)$.

Intuitively, since $\mathcal{P}$ is acyclic, Condition (*ii*) of Definition 4.3 is irrelevant, and the unfoundedness conditions coincide with the conditions of applicability of the $\gamma_{\mathcal{P}}$ operator of Definition 4.2.

The next theorem follows from the results in [Leone et al., 1997].

Theorem 5.2 Let $\mathcal{P}$ be a HCF ground program and I be an interpretation, then

- I belongs to $\mathbf{I}_{\mathcal{P}}$ (i.e., I is in the domain of $\mathcal{W}_{\mathcal{P}}$),
- $\mathcal{W}^{\infty}_{\mathcal{P}}(I)$ is computable in *quadratic time* in the size of $\mathcal{P}$.

6 EFFICIENT COMBINATION OF PRUNING OPERATORS

We now show how to combine the $\Phi_{\mathcal{P}}$ and $\mathcal{W}_{\mathcal{P}}$ operators, resulting in an efficient implementation of DetCons.

6.1 A PONDERED CHOICE

From the previous sections, given a program $\mathcal{P}$ and an interpretation I, we know that:

- the computation of $\Phi^{\infty}_{\mathcal{P}}(I)$ is always very efficient (*linear time computable*);
- $\mathcal{W}_{\mathcal{P}}$ is "stronger" than $\Phi_{\mathcal{P}}$ (i.e., $\Phi_{\mathcal{P}}(I) \subseteq \mathcal{W}_{\mathcal{P}}(I)$ for any interpretation I);
- the computation of $\mathcal{W}_{\mathcal{P}}$ is intractable in the general case (since deciding whether an interpretation belongs to its domain is co-NP-hard);
- the computation of $\mathcal{W}^{\infty}_{\mathcal{P}}(I)$ is tractable (*quadratic*) when $\mathcal{P}$ belongs to the restricted class of *head-cycle-free* programs;
- $\Phi_{\mathcal{P}}$ is equivalent to $\mathcal{W}_{\mathcal{P}}$ when $\mathcal{P}$ belongs to the restricted class of *acyclic* programs.

Based on these observations, we have designed an approach which exploits the positive aspects of both operators, including the efficiency of *Fitting*'s operator wherever we are sure that it is equivalent to the *Well-founded* operator or that the computation of the latter it intractable. On the other hand, our approach takes advantage of the (potentially) stronger pruning of the *Well-founded* operator where feasible. To that end we employ some modularity properties of Disjunctive Logic Programs.

Proposition 6.1 [Leone et al., 1997] Let $\Gamma_{\mathcal{P}} \in \{\Phi_{\mathcal{P}}, \ \mathcal{W}_{\mathcal{P}}\}$, and I be an interpretation for a program $\mathcal{P}$. Then, $\Gamma_{\mathcal{P}}(I) = \bigcup_{C \in Comp(\mathcal{P})} \Gamma_{\mathcal{P}_C}(I)$, where $Comp(\mathcal{P})$ is the set of all components of $\mathcal{P}$, and $\mathcal{P}_C$ denotes the subprogram of component C.

Our implementation of DetCons is sketched in Figure 3. Proposition 6.1 allows us to realize a combined use of the two operators (Fitting and Well-founded) by choosing the most suitable, depending on the syntactical structure of each component. In particular, we apply $\mathcal{W}_{\mathcal{P}}$ on *cyclic and HCF* components, where it is stronger than $\Phi_{\mathcal{P}}$ but efficiently computable. On the other hand, we apply $\Phi_{\mathcal{P}}$ on *acyclic* components, where it is equivalent to $\mathcal{W}_{\mathcal{P}}$ and more efficiently computable, and on *cyclic and not-HCF* components, where $\mathcal{W}_{\mathcal{P}}$ is known to be intractable.

Thus, if the input program is acyclic, we always apply the linear operator $\Phi_{\mathcal{P}}$ without any loss in pruning strength; but if the program is cyclic, we limit the application of $\mathcal{W}_{\mathcal{P}}$ to those components (*cyclic and HCF*) where it can result in a strong pruning of the search space.

```
Procedure DetCons(var I: Interpretation,
                  var inconsistency: bool);
begin
  inconsistency := false;
  for each component C ∈ Comp(P) do
  begin
    switch classOf(C)
      case acyclic:
        ComputeFittingFixpoint(I, inconsistency);
      case cyclic-notHCF:
        ComputeFittingFixpoint(I, inconsistency);
      case cyclic-HCF:
        ComputeWellFoundedFixpoint(I, inconsistency);
    end;
    if inconsistency then break;
  end;
end;
```

Figure 3: Procedure DetCons

6.2 IMPLEMENTATION ISSUES

We conclude this section with some implementation remarks. Due to space constraints, DetCons has been presented in a simplified manner and details concerning the standard propagation routines and further propagation rules have been omitted (see [Faber et al., 1999b, Faber et al., 2001] for details).

At the beginning of the ASP computation, we classify the components of the program w.r.t. acyclicity and head-cycle freeness, since the *DetCons* procedure needs this information. Such a classification is done very efficiently in linear time. We first build the dependency graph $DG_{\mathcal{P}}$ of $\mathcal{P}$ (in linear time); then, we compute the strongly connected components of $DG_{\mathcal{P}}$ applying the linear-time Tarjan algorithm [Tarjan, 1972], and we finally scan the components checking whether they are acyclic or HCF.

To implement the *Well-founded* operator, we have designed an algorithm computing $GUS_{\mathcal{P},C}(I)$ for for an interpretation I and a cyclic HCF component C of a program $\mathcal{P}$.

Recall Definition 4.3 where three conditions account for cases in which a set of atoms cannot be derived. Conditions (i) and (iii) basically correspond to rule satisfaction (w.r.t. I and $I - X$, respectively), while condition (ii) is used to detect positive cycles without foundation. The basic idea is to compute $C - GUS_{\mathcal{P},C}(I)$ by incrementally deriving atoms in C which are "founded", i.e., which do not belong to $GUS_{\mathcal{P},C}(I)$. That is, we build a finite sequence $Y_0, \ldots, Y_n$, where $Y_0 = \emptyset$ and $Y_n = C - GUS_{\mathcal{P},C}(I)$. To this end, we look for rules which do not satisfy any of the three conditions of Definition 4.3 (the conditions are checked w.r.t. X set to Y_i and interpretation I). Once one such a rule r is found, we derive that $H(r) \cap C$ (which is a single atom, since the C is HCF) is "founded", that is, it does not belong to $GUS_{\mathcal{P},C}(I)$, and can be added to Y_{i+1}. The foundedness of an atom may imply the foundedness of further atoms; we proceed until a fixpoint is reached.

At the end of the computation, all atoms in $C - Y_n$ are known to be unfounded, and we set them to false in I. This can result in inconsistency if $I \cap C - Y_n \neq \emptyset$, i.e., if an unfounded atom was set to true in I.

Frequently, all atoms in $GUS_{\mathcal{P},C}(I)$ happen to be already false w.r.t. I, and its computation is completely useless. We would like to identify cases where this condition is recognized without actually computing $GUS_{\mathcal{P},C}(I)$. To this end, at each step of DetCons (Figure 3), we propagate the deterministic consequences over all components by means of the *Fitting* operator [2] and then execute the GUS-computation only on some "selected" components instead of "all" HCF and cyclic components.

In particular, we only call the GUS-computation for components where some atom may have become unfounded by the most recent propagation step. In order to do that, we store some further information during the *Fitting* propagation.

Basically, an atom A can become unfounded if it has lost a "potential support", as some rule r containing A in the head has become satisfied during the last propagation (either the body of r has become false or a head atom of r, different from A, has become true).[3] If no atom of a component C has lost any "potential support", then $GUS_{\mathcal{P},C}(I)$ is unaltered, and its computation is superfluous. To automatically recognize such superfluous computations, when a rule becomes satisfied, we push the component of each each atom that loses its support in a queue. And we eventually launch the GUS-computation only for the components stored in this queue, i.e. only for those cyclic and HCF components in which at least one head atom lost a "potentially supporting rule". This way, we avoid a lot of useless GUS computations.

It is worthwhile noting that we have designed and implemented *linear-time* algorithms for computing both the *Greatest Unfounded Set* (as described above) and the *Fitting* operator. These algorithms use propagation queues and suitable counters like the *Dowling and Gallier* [Dowling and Gallier, 1984, Minoux, 1988] algorithm; we omit the description for space limitations,

[2] In a "localized" way, using a technique à la *Dowling and Gallier*.

[3] A (disjunctive) rule can support only one atom in its head.

full details can be found in [Calimeri, 2001].

7 COMPARISONS AND BENCHMARKS

In order to evaluate our intuitions, we have implemented two new pruning operators, based on the conclusions drawn in the previous section in the ASP system **DLV** [Faber et al., 1999b, Faber et al., 2001]. and experimentally compared the new pruning operators against the original pruning operator employed by **DLV**. Next, we describe the compared methods, the benchmark problems and data, and we finally discuss the results of the experiments.

7.1 OVERVIEW OF THE COMPARED METHODS

We have compared the following three methods in the ASP system **DLV**.

Old. The method originally employed by **DLV**. It always uses the generalized Fitting operator $\Phi_{\mathcal{P}}$ introduced in Section 4.1, which is efficiently computable (a fixpoint is reached in linear time), but does not prune the search space as much as $\mathcal{W}_{\mathcal{P}}$.

ifPoss. Based on the generalized Well-founded operator $\mathcal{W}_{\mathcal{P}}$ introduced in Section 4.2, and exploiting observations from Section 6, this method avoids the use of $\mathcal{W}_{\mathcal{P}}$ on those components where its computation is very expensive (i.e., deciding its applicability is intractable). It employs $\mathcal{W}_{\mathcal{P}}$ on all head-cycle free components, while it resorts to the generalized Fitting operator $\Phi_{\mathcal{P}}$ on the remaining (i.e., non-HCF) components.

ifNeed. This is the method described in Figure 3. It fully implements the theoretical results from Section 6, using both $\Phi_{\mathcal{P}}$ and $\mathcal{W}_{\mathcal{P}}$ where appropriate, and is a refinement of method **ifPoss**. $\mathcal{W}_{\mathcal{P}}$ is only used on cyclic head-cycle free components, whereas $\Phi_{\mathcal{P}}$ is applied for all acyclic components.

7.2 BENCHMARK PROBLEMS

To properly evaluate the pruning techniques described in the previous sections, we chose a couple of benchmark problems: Hamiltonian path, Blocksworld Planning, and Sokoban.

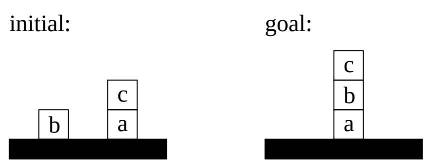
Hamiltonian Path (HAMPATH) is a classical NP-complete problem from graph theory: Given an undirected graph $G = (V, E)$, where V is the set of vertices of G and E is the set of edges, and a node $a \in V$ of this graph, does there exist a path of G starting at a and passing through each node in V exactly once?

Suppose that the graph G is specified by two predicates $node(X)$ and $arc(X, Y)$, and the starting node is specified by the predicate $start$ which contains only a single tuple. Then, the following program solves the problem HAMPATH:

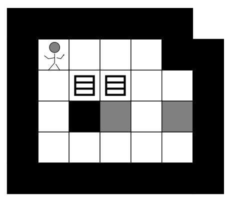
$reached(X)$:- $start(X)$.
$reached(X)$:- $inPath(Y, X)$.
:- $node(X)$, not $reached(X)$.
$inPath(X, Y) \vee outPath(X, Y)$:- $reached(X), arc(X, Y)$.
:- $inPath(X, Y), inPath(X, Y1), Y <> Y1$.
:- $inPath(X, Y), inPath(X1, Y), X <> X1$.

Blocksworld (BW) is a classic problem from the planning domain, and one of the oldest problems in AI: Given a table and a number of blocks in a (known) initial state and a desired goal state, try to reach that goal state by moving one block at a time such that each block is either on top of another block or the table at any given time step.

Due to space restrictions we refer to [Erdem, 1999, Faber et al., 1999a] for a complete encoding. Figure 4 shows a simple instance that can be solved in three time steps: First we move block c to the table, then block b on top of a, and finally c on top of b.



(a) BW Instance



(b) SOKO Instance

Figure 4: BW and SOKO Instances

Sokoban (SOKO) is a game puzzle which has been developed by the Japanese company "Thinking Rabbit, Inc." in 1982. "Sokoban" means "warehouse-keeper" in Japanese. Each puzzle consists of a room layout (a number of square fields representing walls or parts of the floor, some of which are marked as storage space) and a starting situation (one sokoban and a number of boxes, all of which must reside on some floor location). The goal is to move all boxes onto storage locations. To this end, the sokoban can walk on floor locations (unless occupied by some box), and push single boxes onto unoccupied floor locations. Figure 4 shows a typical configuration involving two boxes, where grey fields are storage fields and black fields are walls.

We have written a **DLV** program which finds solutions with a given number of push actions (where one push action can move a box over a number of fields) for a given puzzle together with a script which iteratively runs that **DLV** program with increasing numbers of push actions (starting at one) until some solution is found. In this way solutions with a minimal number of push actions are found.

The puzzle in Figure 4 is solvable with 6 push actions, so the script uses **DLV** to prove that no solutions with 1–5 push actions exist, and then to compute a solution with 6 push actions.

7.3 BENCHMARK DATA

Random graph instances for HAMPATH were generated using a tool[4] by Patrik Simons which has been used to compare Smodels against SAT solvers [Simons, 2000]. For each problem size n we generated *ten* instances, always assuming node 0 as the starting node, and for each instance we stopped after the first solution had been found.

The blocksworld problems P1 to P4 have been employed in [Erdem, 1999] to compare ASP systems, and can be solved in 4, 6, 8 and 9 steps, respectively. We augmented these by problems P5 and P6 which require 11 and 12 steps, respectively. For each of these problems, we generated 8 random permutations of the input. In addition, we also tried to solve each of these problems with one step less than required, which fails to produce any plan but shows the minimality of the regular solutions. These instances are labeled P1-1, P2-1 and so forth in Figure 5.

A vast amount of Sokoban puzzles is available on the Internet in a simple ANSI text format. The examples we used for benchmarks are results of efforts to automatically generate hard puzzles. One set has been created by Yoshio Murase[5], the other set is by Jacques Duthen[6]. The puzzle in Figure 4 is number 2 of Duthen's instances.

[4] `http://tcs.hut.fi/Software/smodels/misc/hamilton.tar.gz` .

7.4 EXPERIMENTAL RESULTS

All experiments were performed on a Pentium III/733 machine with 256MB of main memory running SuSE GNU/Linux (kernel 2.2.14).

For each invocation of **DLV** we allowed a maximum run-time of 600 seconds. For SOKO we have several invocations per problem instance, so the total reported time may be more than 600 seconds. Average running times for HAMPATH and BW are displayed in Figure 5.

	Old	**ifPoss**	**ifNeed**
10	0.02	0.02	0.01
20	0.04	0.04	0.04
30	0.19	0.08	0.07
40	0.11	0.12	0.12
50	-	0.18	0.18
60	-	0.71	0.64
70	-	0.34	0.33
80	-	0.43	0.41
90	-	0.61	0.56
100	-	0.68	0.66
110	-	0.82	0.78
120	-	1.21	1.19

(a) Hamiltonian Path

	Old	**ifPoss**	**ifNeed**
P1 -1	0.02	0.02	0.02
P2 -1	0.04	0.04	0.04
P3 -1	1.58	1.66	1.57
P4 -1	1.26	1.33	1.30
P5 -1	9.48	9.84	9.47
P6 -1	146.92	147.13	146.90
P1	0.02	0.02	0.02
P2	0.06	0.07	0.06
P3	4.11	5.17	4.15
P4	10.56	10.18	10.66
P5	174.21	187.64	174.17
P6	166.12	170.13	166.08

(b) Blocksworld

Figure 5: Average Running Times

On HAMPATH, both **ifPoss** and **ifNeed** perform similarly to **Old** for small problem instances, but scale tremendously better and are able to efficiently deal with graphs of 120 nodes, whereas **Old** cannot solve problems with more than 40 nodes. On average, **ifNeed** slightly outperforms **ifPoss**. These benchmark programs have highly cyclic HCF dependency graphs.

[5] `http://www.ne.jp/asahi/ai/yoshio/sokoban/auto52/auto52.htm` .

[6] `http://hem.passagen.se/awl/ksokoban/sokogen-990602.skm` .

Thus, **ifNeed** and **ifPoss** can exploit the pruning power of the well-founded operator, significantly outperforming **Old** which employs only the Fitting operator. On the other hand, the dependency graphs of these programs usually have one big component containing nearly all atoms. Therefore, there is nearly no difference between **ifNeed** and **ifPoss**, as the former cannot avoid many calls to the well-founded operator.

For BW, **Old** and **ifNeed** are nearly equivalent, and both outperform **ifPoss** by a few percent, though all three approaches seem to scale similarly. We explain this as follows: These programs have only few cyclic HCF components while most components are acyclic. Moreover, these few cyclic components are also very small, and the well-founded operator does not bring a relevant gain in terms of pruning compared to the Fitting operator (so **Old** and **ifNeed** show essentially the same behavior). **ifPoss** pays a computational overhead w.r.t. the other methods, because it is needlessly invoked several times on acyclic components.

SOKO, finally, shows that both **ifPoss** and **ifNeed** perform significantly better than **Old**, which fails to solve more than 50% of all problems instances and usually takes one or two orders of magnitude longer to solve the remaining ones:

	Yoshio Murase		Jacques Duthen	
	solved	unsolved	solved	unsolved
Old	12	40	38	40
ifPoss	41	11	68	10
ifNeed	41	11	69	9

For the Yoshio Murase set, **ifNeed** yields an average speedup of 18.45% over **ifPoss**, the maximum speedup being 60.67% (instance 9). On the Jacques Duthen set, **ifNeed** yields an average speedup of 15% over **ifPoss**, the maximum speedup being 67% (instance 62). A full account of the Sokoban test results is given in a longer version which has been published as a Technical Report [Calimeri et al., 2001].

In summary, our benchmarks show that both **ifPoss** and **ifNeed** are strictly preferable to **Old**, and that of these two, **ifNeed** shows a measurable speedup on a wide range of examples. Therefore the latest **DLV** release employs **ifNeed** by default.

For future work, we plan to focus on (automatically) improving the choice of pruning operators and tuning the actual implementations of these operators. Another perspective for future work would be to evaluate the applicability and impact of even "stronger" operators, e.g. the $\mathcal{F}_S$ operator introduced in [Baral, 1992].

Yet another issue to be considered is the fact that dependency graphs in **DLV** are currently static. They could be made dynamic in the sense that only currently undefined atoms are considered in the graphs. This could have the advantage that non-HCF components could become HCF and cyclic components could become acyclic during the computation, allowing for more efficient algorithms on these components. But it is unclear how the computational cost of maintaining these graphs relates to possible savings.

Acknowledgments

This work was supported by the Austrian Science Fund (FWF) under grants P14781-INF and Z29-INF, and by the European Commission under project ICONS, project no. IST-2001-32429.

References

[Anger et al., 2001] Anger, C., Konczak, K., and Linke, T. (2001). `NoMoRe`: A System for Non-Monotonic Reasoning. *LPNMR'01*, number 2173 in Lecture Notes in AI (LNAI), pp. 406–410.

[Baral, 1992] Baral, C. (1992). Generalized Negation As Failure and Semantics of Normal Disjunctive Logic Programs. *LPAR'92*, pp. 309–319.

[Bell et al., 1994] Bell, C., Nerode, A., Ng, R. T., and Subrahmanian, V. (1994). Mixed Integer Programming Methods for Computing Nonmonotonic Deductive Databases. *JACM*, 41:1178–1215.

[Ben-Eliyahu and Dechter, 1994] Ben-Eliyahu, R. and Dechter, R. (1994). Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87.

[Calimeri, 2001] Calimeri, F. (2001). Progettazione e sviluppo di tecniche di ottimizzazione per sistemi di basi di conoscenza. Master's thesis, D.E.I.S., Università degli Studi della Calabria, Rende (CS), Italy.

[Calimeri et al., 2001] Calimeri, F., Faber, W., Leone, N., and Pfeifer, G. (2001). Pruning Operators for Answer Set Programming Systems. Technical Report INFSYS RR-1843-01-07, Institut für Informationssysteme, Technische Universität Wien, Austria.

[Dowling and Gallier, 1984] Dowling, W. F. and Gallier, J. H. (1984). Linear-time algorithms for testing the satisfability of propositional horn formulae. *Journal of Logic Programming*, 3:267–284.

[East and Truszczyński, 2000] East, D. and Truszczyński, M. (2000). dcs: An implementation of DATALOG with Constraints. *NMR'2000*, Breckenridge, Colorado, USA.

[Egly et al., 2000] Egly, U., Eiter, T., Tompits, H., and Woltran, S. (2000). Solving Advanced Reasoning Tasks using Quantified Boolean Formulas. In *AAAI'00*, pp. 417–422. AAAI Press / MIT Press.

[Eiter et al., 1997] Eiter, T., Gottlob, G., and Mannila, H. (1997). Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418.

[Erdem, 1999] Erdem, E. (1999). Applications of Logic Programming to Planning: Computational Experiments. Unpublished draft. `http://www.cs.utexas.edu/users/esra/papers.html`.

[Faber et al., 1999a] Faber, W., Leone, N., Mateis, C., and Pfeifer, G. (1999a). Using Database Optimization Techniques for Nonmonotonic Reasoning. *DDLP'99*, pp. 135–139. Prolog Association of Japan.

[Faber et al., 1999b] Faber, W., Leone, N., and Pfeifer, G. (1999b). Pushing Goal Derivation in DLP Computations. *LPNMR'99*, number 1730 in Lecture Notes in AI (LNAI), pp. 177–191.

[Faber et al., 2001] Faber, W., Leone, N., and Pfeifer, G. (2001). Experimenting with Heuristics for Answer Set Programming. In *IJCAI 2001*, pp. 635–640, Seattle, WA, USA. Morgan Kaufmann Publishers.

[Fages, 1994] Fages, F. (1994). Consistency of clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1(1):51–60.

[Fitting, 1985] Fitting, M. (1985). A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312.

[Gelfond and Lifschitz, 1991] Gelfond, M. and Lifschitz, V. (1991). Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385.

[Leone et al., 1993] Leone, N., Romeo, M., Rullo, P., and Saccà, D. (1993). Effective implementation of negation in database logic query languages. In *LOGIDATA+: Deductive Database with Complex Objects*, LNCS 701, pp. 159–175.

[Leone et al., 1997] Leone, N., Rullo, P., and Scarcello, F. (1997). Disjunctive stable models: Unfounded sets, fixpoint semantics and computation. *Information and Computation*, 135(2):69–112.

[Lifschitz, 1996] Lifschitz, V. (1996). Foundations of Logic Programming. *Principles of Knowledge Representation*, pp. 69–127. CSLI Publications, Stanford.

[Lifschitz and Turner, 1994] Lifschitz, V. and Turner, H. (1994). Splitting a Logic Program. *ICLP'94*, pp. 23–37. MIT Press.

[McCain and Turner, 1998] McCain, N. and Turner, H. (1998). Satisfiability Planning with Causal Theories. *KR'98*, pp. 212–223. Morgan Kaufmann Publishers.

[Minoux, 1988] Minoux, M. (1988). LTUR: a simplified linear-time unit resolution algorithm for Horn formulae and computer implementation. *Information Processing Letters*, 29:1–12.

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

[Niemelä and Simons, 1996] Niemelä, I. and Simons, P. (1996). Efficient Implementation of the Well-founded and Stable Model Semantics. *ICLP'96*, pp. 289–303, Bonn, Germany. MIT Press.

[Rao et al., 1997] Rao, P., Sagonas, K. F., Swift, T., Warren, D. S., and Freire, J. (1997). XSB: A System for Efficiently Computing Well-Founded Semantics. *LPNMR'97*, number 1265 in Lecture Notes in AI (LNAI), pp. 2–17, Dagstuhl, Germany.

[Scarcello, 1997] Scarcello, F. (1997). *Caratterizzazione e Calcolo di Modelli Stabili in Programmazione Logica Disgiuntiva*. PhD thesis, Università della Calabria, Rende (CS), Italy.

[Simons, 2000] Simons, P. (2000). *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, Finland.

[Subrahmanian et al., 1995] Subrahmanian, V., Nau, D., and Vago, C. (1995). WFS + Branch and Bound = Stable Models. *IEEE TKDE*, 7(3):362–377.

[Tarjan, 1972] Tarjan, R. (1972). Depth-first search and linear graph algorithm. *SIAM Journal of Computing*, 1(2).

[van Gelder et al., 1991] van Gelder, A., Ross, K., and Schlipf, J. (1991). The Well-Founded Semantics for General Logic Programs. *JACM*, 38(3):620–650.