A Backjumping Technique for Disjunctive Logic Programming

Wolfgang Faber, Nicola Leone, and Francesco Ricca

Department of Mathematics
University of Calabria
87030 Rende (CS), Italy
{faber,leone,ricca}@mat.unical.it

Abstract. In this work we present a backjumping technique for Disjunctive Logic Programming (DLP) under the Answer Set Semantics. It builds upon related techniques that had originally been proposed for propositional satisfiability testing, which have been adapted to non-disjunctive Answer Set Programming (ASP) recently [1, 2].
We focus on backjumping without clause learning. We provide a new theoretical framework for backjumping on Disjunctive Logic Programs. We optimize the reason calculus, reducing the information to be stored, while fully preserving the correctness and the efficiency of the jumping technique. We implement the proposed technique in DLV, the state-of-the-art DLP system. We have conducted several experiments on hard random problems in order to assess the impact of backjumping. Our conclusion is that when lookahead is employed, there is basically no advantage when enabling backjumping. However, when lookahead is disabled, we can observe that the number of choices in general decreases by a non-negligible factor. In our (naive) implementation this gain is (often more than) compensated by the additional overhead incurred by the reason calculus. It is unclear whether one can reduce this overhead by a more efficient implementation. We therefore conjecture that, at least on hard unstructured instances, backjumping only has an impact when lookahead is not active and when clause learning is employed in addition.

1 Introduction

Answer Set Programming (ASP) in its general form allows for disjunction in rule heads and nonmonotonic negation in rule bodies. This knowledge representation language is very expressive in a precise mathematical sense: *Every* problem in the complexity class Σ_2^P and Π_2^P (under brave and cautious reasoning, respectively) can be expressed [3]. Thus, ASP is strictly more powerful than SAT-based programming, as it allows us to solve problems which cannot be translated to SAT in polynomial time. The high expressive power of ASP can be profitably exploited in AI, which often has to deal with problems of this complexity. For instance, several problems in diagnosis and planning under incomplete knowledge are complete for the complexity class Σ_2^P or Π_2^P [4, 5], and can be naturally encoded in ASP [6, 7].

Most of the optimization work on ASP systems has focused on the efficient evaluation of non-disjunctive programs (whose power is limited to NP/co-NP), whereas the optimization of full (disjunctive) ASP programs has been treated in fewer works (e.g., in [8, 9]).

One of the more recent proposals for enhancing the evaluation of non-disjunctive programs has been the definition of backjumping and clause learning mechanisms. These techniques had been successfully employed in propositional SAT solvers before, and were "ported" to non-disjunctive ASP in [1, 2], resulting in the system Smodels$_{cc}$.

In this paper we address two issues:

▶ A generalization of backjumping to disjunctive programs.
▶ Is backjumping without clause learning effective?

We first present a generalization of the work in [1, 2] to disjunctive programs by defining a *reason calculus* for the DetCons function of DLV (which roughly corresponds to unit propagation in DPLL-based SAT solvers and AtLeast/AtMost in Smodels). These reasons allow for effective backjumping. We also describe the implementation of the reason calculus in the DLV system, the state-of-the-art disjunctive ASP system. In fact, our implementation aims at reducing the information to be stored as much as possible, while maintaining the best jumping possibilities.

Subsequently, we assess our method and implementation by experimentation on hard, randomly generated instances. We observe several issues: 1. In conjunction with lookahead, there is basically no advantage of backjumping. 2. Without lookahead, we observe that the number of choices often decreases by a non-negligible factor. 3. However, the time needed for maintaining the information for the reason calculus apparently supersedes the gain of having fewer choices. It is unclear whether this is because of our unoptimized implementation or a general problem.

Summarizing, we observe that backjumping without clause learning is not effective (at least on unstructured instances), unless one is able to find a highly optimized implementation of the reason calculus. Since this task appears to be difficult to achieve, we conjecture (based on the results of [1]) that backjumping should be combined with clause learning.

2 Preliminaries on Disjunctive Logic Programming

In this section, we provide a brief introduction to the syntax and semantics of Disjunctive Logic Programming; for further background see [10, 11].

2.1 Syntax

A *(disjunctive) rule* r is a formula

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

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are atoms[1] 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

[1] For simplicity, we do not consider strong negation in this paper. It can be emulated by introducing new atoms and integrity constraints.

$B^-(r) = \{\text{not } b_{k+1}, ..., \text{not } b_m\}$ 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.

A *disjunctive logic 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 $\text{not}.l = a$ if $l = \text{not } a$, otherwise $\text{not}.l = \text{not } l$, and given a set L of literals, $\text{not}.L = \{\text{not}.l \mid l \in L\}$.

2.2 Semantics

The semantics of a disjunctive logic program is given by its (consistent) answer sets [11]; on the language considered here these are equal to disjunctive stable models of [12].

Given a program $\mathcal{P}$, let the *Herbrand Universe* $U_{\mathcal{P}}$ be the set of all constants appearing in $\mathcal{P}$ and the *Herbrand Base* $B_{\mathcal{P}}$ be the set of all possible ground atoms which can be constructed from the predicate symbols appearing in $\mathcal{P}$ with the constants of $U_{\mathcal{P}}$.

Given a rule r, $Ground(r)$ denotes the set of rules obtained by applying all possible substitutions σ from the variables in r to elements of $U_{\mathcal{P}}$. Similarly, given a program $\mathcal{P}$, the *ground instantiation* $\mathcal{P}$ of $\mathcal{P}$ is the set $\bigcup_{r \in \mathcal{P}} Ground(r)$.

For every program $\mathcal{P}$, we define its answer sets using its ground instantiation $\mathcal{P}$ in two steps: First we define the answer sets of positive programs, then we give a reduction of general programs to positive ones and use this reduction to define answer sets of general programs.

A set L of ground literals is said to be *consistent* if, for every atom $\ell \in L$, its complementary literal $\text{not } \ell$ is not contained in L. An interpretation I for $\mathcal{P}$ is a consistent set of ground literals over atoms in $B_{\mathcal{P}}$.[2] A ground literal ℓ is *true* w.r.t. I if $\ell \in I$; ℓ is *false* w.r.t. I if its complementary literal is in I; ℓ is *undefined* w.r.t. I if it is neither true nor false w.r.t. I.

Let r be a ground rule in $\mathcal{P}$. The head of r is *true* w.r.t. I if exists $a \in H(r)$ s.t. a is true w.r.t. I (i.e., some atom in $H(r)$ is true w.r.t. I). The body of r is *true* w.r.t. I if $\forall \ell \in B(r)$, ℓ is true w.r.t. I (i.e. all literals on $B(r)$ are true w.r.t I). The body of r is *false* w.r.t. I if $\exists \ell \in B(r)$ s.t. ℓ is false w.r.t I (i.e., some literal in $B(r)$ is false w.r.t. I). The rule r is *satisfied* (or *true*) w.r.t. I if its head is true w.r.t. I or its body is false w.r.t. I.

Interpretation I is *total* if, for each atom A in $B_{\mathcal{P}}$, either A or $\text{not}.A$ is in I (i.e., no atom in $B_{\mathcal{P}}$ is undefined w.r.t. I). A total interpretation M is a *model* for $\mathcal{P}$ if, for every $r \in \mathcal{P}$, at least one literal in the head is true w.r.t. M whenever all literals in the body are true w.r.t. M. X is an *answer set* for a positive program $\mathcal{P}$ if its positive part is minimal w.r.t. set inclusion among the models of $\mathcal{P}$.

The *reduct* or *Gelfond-Lifschitz transform* of a general ground program $\mathcal{P}$ w.r.t. an interpretation X is the positive ground program $\mathcal{P}^X$, obtained from $\mathcal{P}$ by (i) deleting

[2] We represent interpretations as sets of literals, since we have to deal with partial interpretations in the next sections.

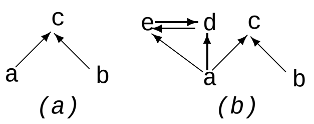

Fig. 1. Graphs (a) $DG_{\mathcal{P}_4}$, and (b) $DG_{\mathcal{P}_5}$

all rules $r \in \mathcal{P}$ whose negative body is false w.r.t. X and (ii) deleting the negative body from the remaining rules.

An answer set of a general program $\mathcal{P}$ is a model X of $\mathcal{P}$ such that X is an answer set of $\mathcal{P}^X$.

2.3 Some ASP properties

Given an interpretation I for a ground program $\mathcal{P}$,we say that a ground atom A is *supported* in I if there exists a *supporting* rule $r \in ground(\mathcal{P})$ such that the body of r is true w.r.t. I and A is the only true atom in the head of r. If M is an answer set of a program $\mathcal{P}$, then all atoms in M are supported [13–15].

An important property of answer sets is related to the notion of *unfounded set* [16, 14]. Let I be a (partial) interpretation for a ground 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) $B(r) \cap \text{not}.I \neq \emptyset$, (ii) $B^+(r) \cap X \neq \emptyset$, (iii) $(H(r) - X) \cap I \neq \emptyset$.

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[3]. 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.

If M is a total interpretation for a program $\mathcal{P}$. M is an answer set of $\mathcal{P}$ iff $\text{not}.M = GUS_{\mathcal{P}}(I)$ [14].

With every ground program $\mathcal{P}$, we associate a directed graph $DG_{\mathcal{P}} = (N, E)$, called the *dependency graph* of $\mathcal{P}$, in which (i) each atom of $\mathcal{P}$ is a node in N and (ii) there is an arc in E directed from a node a to a node b iff there is a rule r in $\mathcal{P}$ such that b and a appear in the head and body of r, respectively.

The graph $DG_{\mathcal{P}}$ singles out the dependencies of the head atoms of a rule r from the positive atoms in its body.[4]

Example 1. Consider the program $\mathcal{P}_4 = \{a \vee b.\ ;\ c \text{ :- } a.\ ;\ c \text{ :- } b.\}$*, and the program* $\mathcal{P}_5 = \mathcal{P}_4 \cup \{d \vee e \text{ :- } a.\ ;\ d \text{ :- } e.\ ;\ e \text{ :- } d, \text{not } b.\}$*. The dependency graph* $DG_{\mathcal{P}_4}$ *of* $\mathcal{P}_4$ *is depicted in Figure 1 (a), while the dependency graph* $DG_{\mathcal{P}_5}$ *of* $\mathcal{P}_5$ *is depicted in Figure 1 (b).*

A program $\mathcal{P}$ is *head-cycle-free* (*HCF*) iff there is no rule$r$ in $\mathcal{P}$ such that two atoms occurring in the head of r are in the same cycle of $DG_{\mathcal{P}}$ [17].

[3] While for non-disjunctive programs the union of unfounded sets is an unfounded set for all interpretations, this does not hold for disjunctive programs (see [14]).

[4] Note that negative literals cause no arc in $DG_{\mathcal{P}}$.

Example 2. The dependency graphs given in Figure 1 reveal that program $\mathcal{P}_4$ of Example 1 is HCF and that program $\mathcal{P}_5$ is not HCF, as rule $d \vee e \leftarrow a$ contains in its head two atoms belonging to the same cycle of $DG_{\mathcal{P}_5}$.

A *component* C of a dependency graph DG is a maximal subgraph of DG such that each node in C is reachable from any other. The *subprogram* of C consists of all rules having some atom from C in the head. An atom is non-HCF if the subprogram of its component is non-HCF. The *subprogram for a component* consists of all rules having a head atom in the component. An atom is non-HCF if it occurs in a non-HCF component.

3 Model Generation in DLV

In this section, we briefly describe the computational process performed by the DLV system [14, 18] to compute answer sets, which will be used for the experiments. Note that, other ASP systems like Smodels [19, 20] 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 $\mathcal{P}$ [21]. The nondeterministic part of the computation is then performed on this simplified ground program by the Model Generator, which is sketched below. Note that for reasons of presentation, the description here is quite simplified; in particular, the choicepoints and search trees are somewhat more complex in the "real" implementation. However, one can find a one-to-one mapping to the simpler formalism described here. A more detailed description can be found in [18]. Note also that the version described here computes one answer set for simplicity, however modifying it to compute all or n answer sets is straightforward. For brevity, $\mathcal{P}$ refers to the simplified ground program in the sequel.

```
bool MG ( Interpretation& I ) {
    if ( ! DetCons ( I ) ) then return false;
    if ( "no atom is undefined in I" ) then return IsAnswerSet(I);
    Select an undefined atom $A$ using a heuristic;
    if ( MG ( $I \cup \{A\}$ ) then return true;
      else return MG ( $I \cup \{\text{not } A\}$ ); };
```

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. Initially, the MG 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).

DetCons(I) computes the deterministic consequences of I, and will be described in more detail in the sequel. If DetCons(I) does not detect any inconsistency, an atom A is selected according to a heuristic criterion and MG is recursively called on both $I \cup \{A\}$ and $I \cup \{\text{not } A\}$. The atom A corresponds to a *branching variable* in SAT solvers.

The efficiency of the whole process depends on two crucial features: a good heuristic to choose the branching variables and an efficient implementation of DetCons. Actually, the DLV system implements a so called *lookahead* heuristic [22, 23] and an efficient DetCons implementation [24, 25].

It is worth noting that, if during the execution of the MG function a contradiction arises, or the answer set candidate is not a minimal model, MG backtracks and modifies the last choice. This kind of backtracking is called chronological backtracking.

In the following sections, we describe a technique in which the truth value assignments causing a conflict are identified and backtracking is performed "jumping" directly to a point so that at least one of those assignments is modified. This kind of backtracking technique is called non-chronological backtracking or backjumping.

4 Backjumping

In this section we first motivate by means of an example how a backjumping technique is supposed to work, and then give a more formal account on how to extend the functions DetCons and MG of DLV to accomplish this task in general. Since the functionality of DetCons is crucial for this task, we start by describing a simplification of it.

4.1 DetCons

As previously pointed out, the role of DetCons is similar to the Boolean Constraint Propagation (BCP) procedure in Davis-Putnam SAT solvers. However, DetCons is more complex than BCP, which is based on the simple unit propagation inference rule, while DetCons implements a set of of inference rules. Those rules combine an extension of the Well-founded operator for disjunctive programs with a number of techniques based on ASP program properties [24, 25].

While the full implementation of DetCons involves four truthvalues (apart from true, false, and undefined, there is also "must be true"), we treat "must be true" as true in this description, as they are treated in the same way with respect to backjumping. Moreover, we group the inference rules using the same terminology as [1] for better comparability:

1. Forward Inference,
2. Kripke-Kleene negation,
3. Contraposition for true heads,
4. Contraposition for false heads,
5. Well-founded negation.

Rule 1 derives an atom as true if it occurs in the head of a rule in which all other head atoms are false and the body is true. Rule 2 derives an atom as false if no rule can support it. Rule 3 applies if for a true atom only one rule that can support it is

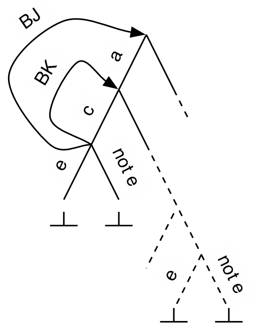

Fig. 2. Backtracking vs Backjumping.

left, and makes inferences such that the rule can support the atom, i.e. derives all other head atoms as false, atoms in the positive body as true and atoms in the negative body as false. Rule 4 makes inferences for rules which have a false head: If only one body literal is undefined, derive a truth value for it such that the body becomes false. Finally, rule 5 sets all members of the greatest unfounded set to false. We note that rule 5 is only applied on recursive HCF subprograms for complexity reasons [25]

4.2 Backjumping by Example

Consider the following program

$$r_1: \ a \vee b. \quad r_2: \ c \vee d. \quad r_3: \ e \vee f.$$
$$r_4: \ g \text{ :- } a, e. \quad r_5: \ \text{ :- } g, a, e. \quad r_6: \ g \text{ :- } a, f. \quad r_7: \ \text{ :- } g, a, f.$$

and suppose that the search tree is as depicted in Fig. 2.

According to this tree, we first assume a to be true, deriving b to be false (because of r_1 and rule 3). Then we assume c to be true, deriving d to be false (because of r_2 and rule 3). Third, we assume e to be true and derive f to be false (because of r_3 and rule 3) and g to be true (because of r_4 and rule 1). This truth assignment violates constraint r_5 (because rule 4 derives g to be false), yielding an inconsistency. We continue the search by inverting the last choice, that is, we assume e to be false and we derive f to be true (because of r_3 and rule 1) and g to be true (because of r_7 and rule 1), but obtain another inconsistency (because of constraint r_7 and rule 4, g must also be false).

At this point, MG goes back to the previous choicepoint, in this case inverting the truth value of c (cf. the arc labelled BK in Fig. 2).

Now it is important to note that the inconsistencies obtained are independent of the choice of c, and only the truth value of a and e are the “reasons” for the encountered inconsistencies. In fact, no matter what the truth value of c is, if a is true then any truth assignment for e will lead to an inconsistency. Looking at Fig. 2, this means that in the whole subtree below the arc labelled a no answer set can be found. It is therefore obvious that the chronological backtracking search explores branches of the search tree that cannot contain an answer set, performing a lot of useless work.

A better policy would be to go back directly to the point at which we assumed a to be true (see the arc labelled BJ in Fig. 2). In other words, if we know the "reasons" of an inconsistency, we can backjump directly to the closest choice that caused the inconsistent subtree.

4.3 Reasons for Inconsistencies

Until now, we used the term "reason" of an inconsistency in an intuitive way. We will now define more formally what such reasons are and how they can be handled.

We start by describing the intuition of "reason" of a ground literal (representing a truthvalue of the literal's atom). A rule a :- $b, c, notd.$ intuitively means: If b and c are true and d is false in the current partial interpretation, then derive a to be true. We can equivalently say that: a is true "because" b and c are true and d is false. In other words, the reasons of the truth value of a derived ground literal are the reasons of literals that entail its truth. For "chosen" literals, their only reason is their choice.

We define the reason of a literal x in the following way: (i) if x is a chosen literal (a "branching variable") then the reason of x is x. (ii) if x is a derived literal, then the reason of x is the union of the reasons of literals that entail its truth. Note that the reason of a literal basically is the set of chosen literals that entail its truth.

Now we are able to define the reason of an inconsistency. An inconsistency occurs when l is determined to be both true and false. Therefore the reason of an inconsistency is the union of the reasons of l and $\text{not } l$. Referring to the example in Section 4.2, the reason of the first inconsistency is the union of the reasons for g and $\text{not } g$, which is the set $R = \{a, e\}$. This means that a and e "entail" this inconsistency.

When, during the search, we assumed a literal l both true and false (and we found, in both cases, an inconsistency) we have two inconsistency reasons, the first one, $R(l)$, is found assuming l true, and the second one, $R(\text{not } l)$, is found assuming l false. It is straightforward to see that to avoid such inconsistencies we have to change one assumption in $R(l) \cup R(\text{not } l)$ which is different from l. It is worth to note that, in order to obtain a complete search we have to change the assumption "nearest" (chronologically) to l (in $R(l) \cup R(\text{not } l)$). If such an assumption does not exist the problem is unsatisfiable (there is no way to avoid that inconsistency). In the example of Section 4.2, l is e, and $R(e) \cup R(\text{not } e) = \{a\}$.

4.4 Extending DetCons with the Reason Calculus

In this section we describe how to compute the reason of a derived literal. In particular, we will modify DetCons to compute the reasons of the derived literals in order to allow for backjumping.

First of all, we introduce a numeric representation of reasons based on the recursion level of the search algorithm (corresponding to the depth of the search tree). We associate an integer number (starting from 0), representing the current recursion level, to each literal which has been chosen during the computation. Each literal l derived during the propagation (through DetCons) will have an associated set of positive integers $R(l)$ representing the reason of l, which stand for the set of choices entailing l. For instance, if $R(a) = \{1, 3, 4\}$, then the literals chosen at recursion level 1,3 and 4 entail a.

Moreover, given a rule r, a ”satisfying literal” is either a true head atom or a false body literal in r. Rule r is satisfied if it has a satisfying literal.

We now define an ordering $\prec$ among satisfying literals of a given rule, which is basically a lexicographic order over the numerically ordered reasons of the literals. We first give two technical definitions: $R_k(l)$ denotes the set of reasons without the k greatest reasons, and $MAX_k(l)$ gives the k-th reason in descending order (or -1 if less than k reasons exist).

$$R_k(l) = \begin{cases} R(l), & k = 1 \\ R_{k-1}(l) \backslash \{max(R_{k-1}(l))\}, & k > 1 \end{cases}$$
$$MAX_k(l) = \begin{cases} max(R_k(l)), & R_k(l) \neq \emptyset \\ -1, & otherwise. \end{cases}$$

where $max(x)$ is the maximum element in the set x. If s_1, s_2 are satisfying literals for rule r, then $s_1 \prec s_2$ (s_1 precedes s_2) iff one the following conditions holds:

(i) $MAX_1(s_1) < MAX_1(s_2) \wedge MAX_1(s_1) > 0 \wedge MAX_1(s_2) > 0$
(ii) $MAX_1(s_1) = -1 \wedge MAX_1(s_2) > 0$
(iii) $\exists k : k > 1 : \forall x : 1 < x < k : MAX_x(s_1) = MAX_x(s_2) \neq -1$ and $MAX_k(s_1) < MAX_k(s_2) \wedge MAX_k(s_1) > 0 \wedge MAX_k(s_2) > 0$
(iv) $\exists k : k > 1 : \forall x : 1 < x < k : MAX_x(s_1) = MAX_x(s_2) \neq -1$ and $MAX_k(s_1) = -1 \wedge MAX_k(s_2) > 0$

Let $s_1 \sim s_2$ if $s_1 \not\prec s_2$ and $s_2 \not\prec s_1$. We write $s_1 \preceq s_2$ iff $s_1 \prec s_2$ or $s_1 \sim s_2$.

Let be s_1,...s_n satisfying literals for rule r, if $s_i \preceq s_j$ for each $j = 1, ..., n$ then $R_r = R(s_i)$ is a *cancelling assignment* for r. Note that, the cancelling assignment of a rule r represents the “earliest” reason causing r to be satisfied.

In the following, we describe how reasons of derived literals are computed for the respective inference rules of DetCons with respect to a partial interpretation I.

Forward Inference
Given a rule r, if $\exists a_i \in H(r)$ such that (i) $\forall b \in B(r)$, $b \in I$, (ii) $\forall a \in (H(r) \setminus \{a_i\})$, $\text{not}\ a \in I$, then infer a_i. The reason of a_i is set to $R(a_i) = \bigcup_{a \in H(r)\setminus\{a_i\}} R(\text{not}a) \cup \bigcup_{l \in B(r)} R(l)$, which is the union of the reasons of all head atoms apart from a_i to be false, and the reasons for the body literals to be true. For example, given the following program:

$$r_1 : \ a \vee b \text{ :- } c, \text{not}\ d. \quad r_2 : \ c \text{ :- } \text{not}\ d, e. \quad r_3 : \ f \vee b. \quad r_4 : \ g \vee d. \quad r_5 : \ e \vee h.$$

Suppose $I = \{\text{not}\ b, c, \text{not}\ d, f, g, \text{not}\ h\}$, and $R(f) = R(\text{not}\ b) = \{1\}$, $R(c) = \{2, 3\}$, $R(g) = R(\text{not}\ d) = \{2\}$, and $R(e) = R(\text{not}\ h) = \{3\}$. We have that a is derived to be true from rule r_1 (all body literals are true and the only head atom b is false) and the reason of a to be true is set to $R(a) = R(\text{not}b) \cup R(c) \cup R(\text{not}d) = \{1, 2, 3\}$.

Kripke-Kleene negation
Given an atom a, if for each rule r such that $a \in H(r)$ (i) $\exists b \in B(r)$ such that $\text{not}.b \in I$ or (ii) $\exists c \in H(r)$ such that $a \neq c$ and $c \in I$, then infer $\text{not}.a$. We then set $R(a) = \bigcup_{r : a \in H(r)} R_r$, where R_r is a cancelling assignment of rule r. So, a becomes false

because each rule with a in the head is cancelled. For example, consider the following subprogram:

$$r_1: \ a \lor b \text{ :- } c, \text{not } d. \quad r_2: \ b \text{ :- } e, \text{not } f. \quad r_3: \ b \text{ :- } g, h.$$

Suppose $I = \{a, c, d, e, f, g, \text{not } h\}$ and $R(a) = \{7\}$, $R(c) = \{5\}$, $R(d) = \{6\}$, $R(e) = \{3\}$, $R(f) = \{4\}$, $R(g) = \{1\}$, $R(\text{not } h) = \{2\}$. The atom b is undefined and it is contained in the head of all rules. The cancelling assignments are as follows: $R_{r_1} = R(c) = \{5\}$, $R_{r_2} = R(e) = \{3\}$, and $R_{r_3} = R(g) = \{1\}$. DetCons then infers b to be false and $R(\text{not } b) = R_{r_1} \cup R_{r_2} \cup R_{r_3} = \{1, 3, 5\}$.

Contraposition for true heads
Given an atom $a \in I$ and a rule r such that $a \in H(r)$, if for each rule $r' \neq r$ such that $a \in H(r')$ (i) $\exists b' \in B(r')$: $\text{not}.b' \in I$ or (ii) $\exists c' \in H(r')$: $c' \neq a \land c' \in I$, then for each $c \in H(r)$ s.t. $c \neq a$ infer $\text{not}.c$, and for each $b \in B(r)$ infer b.

For each rule r' with a in the head (different from r) take a cancelling assignment $R_{r'}$ and set, for each derived literal l of r, $R(l) = R(a) \cup \bigcup_{r': a \in H(r') \land r' \neq r} R_{r'}$. So the reason for the derived literals is the reason for a and the reasons for all other rules with a in the head to be cancelled. For example, consider the following subprogram:

$$r_1: \ a \lor b \text{ :- } c, \text{not } d. \quad r_2: \ a \lor g \text{ :- } f. \quad r_3: \ a \text{ :- } k.$$

Suppose $I = \{a, f, g, \text{not } k\}$ and $R(a) = \{2\}$, $R(f) = \{2\}$, $R(g) = \{3\}$, $R(\text{not } k) = \{1\}$. The only unsatisfied rule having a in the head is r_1 and the cancelling assignments are $R_{r_2} = R(a) = R(f) = \{2\}$ and $R_{r_3} = R(\text{not } k) = \{1\}$. In this case we infer c and $\text{not } d$ and $\text{not}b$ and set $R(\text{not } b) = R(c) = R(\text{not } d) = R(a) \cup R_{r_2} \cup R_{r_3} = \{1, 2\}$.

Contraposition for false heads
Given a rule r such that (i) $\forall a \in H(r) : \text{not}.a \in I$, (ii) $\exists l \in B(r) : l \notin I \land \text{not}.l \notin I$, (iii) $\forall b \in B(r) \setminus \{l\} : b \in I$, then infer $\text{not}.l$. We set $R(l) = \bigcup_{a \in H(r)} R(\text{not}a) \cup \bigcup_{b \in B(r) \setminus \{l\}} R(b)$, so the reason for l is the union of reasons for the head atoms to be false and the reasons for the body literals (apart from l) to be true. Consider the following subprogram:

$$r_1: \ a \lor b \text{ :- } c, d. \quad r_2: \ d \lor a \lor b. \quad r_3: \ e \lor a. \quad r_2: \ \text{ :- } d, b.$$

Suppose $I = \{\text{not } a, \text{not } b, d, e\}$, and $R(\text{not } a) = \{1\}$, $R(\text{not } b) = R(d) = \{3\}$, $R(e) = \{2\}$. By r_1, we get $\text{not } c$ and $R(\text{not } c) = R(\text{not } a) \cup R(\text{not } b) \cup R(d) = \{1, 3\}$.

Well-founded negation
Let S be an HCF subprogram of P, I be an unfounded-free interpretation, and X be the greatest unfounded set of S w.r.t. I. Then infer all atoms in X to be false. For each atom $a \in X$ and for each rule r with a in the head, set $R(a) = \bigcup_{r \in S: a \in H(r)} R_r^*$, where R_r^* is the cancelling assignment of r, if r is satisfied w.r.t. I, or $R_r^* = \emptyset$ if r is not satisfied w.r.t. I (in the latter case r contains some element from X).

In other words, if an atom a_i is unfounded we derive it as false. This case, a_i becomes false because each rule r_i, with a_i in the head, has been cancelled or a_i is supported by an unfounded atom. For example, consider the following program:

$$r_1: \ a \lor b. \quad r_2: \ a \text{ :- } \text{not } c. \quad r_3: \ a \text{ :- } d. \quad r_4: \ d \text{ :- } a. \quad r_5: \ b \lor e. \quad r_6: \ c \lor f.$$

Suppose $I = \{b, c, \text{not } e, \text{not } f\}$, $R(b) = R(\text{not } e) = \{1\}, R(c) = R(\text{not } f) = \{2\}$. The greatest unfounded set is $X = \{a, d\}$, then infer a and d to be false and set $R(\text{not } a) = R(\text{not } d) = R^*_{r_1} \cup R^*_{r_2} \cup R^*_{r_3} \cup R^*_{r_4} = \{1, 2\}$, where $R^*_{r_1} = R(b) = \{1\}$, $R^*_{r_2} = R(c) = \{2\}$ and $R^*_{r_3} = R^*_{r_4} = \emptyset$.

Inconsistency reasons In general, an inconsistency occurs if a literal l should be true and false in the same interpretation. We call such an l a *conflicting literal*. The reason of the detected inconsistency is the set $C = R(l) \cup R(\text{not}.l)$ (the union of the reason of l to be true and l to be false). This computation is independent of the inference rule used during the propagation to determine the inconsistencies.

4.5 Model Generator with backjumping

In this section we describe MGBJ (shown in Fig. 3), a modification of the MG function (as described in section 3), which is able to perform non-chronological backtracking. It extends MG by introducing additional data structures, in order to keep track of reasons and to control backtracking. In particular, two new variables bj_level and $curr\_level$ are used to store the current level of recursion and to control backtracking, respectively. bj_level represents the level to which we (back)jump at the end of each MGBJ call. In addition, inconsistency reasons are stored in the activation stack by means of the $callerIR$ parameter.

Initially, the MGBJ function is invoked with I and $callerIR$ set to the empty interpretation, and bj_level set to zero (decision levels, corresponding to MGBJ calls, are counted by using nonnegative integers). Like the MG function, 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. Again, it is easy to modify this procedure in order to obtain all or at most n answer sets.

MGBJ first calls the enhanced version of the DetCons procedure, referred to as DetConsH. DetConsH extends the current interpretation with those literals that can be deterministically inferred, and, at the same time, DetConsH computes their "reasons" as described in Section 4.4. In particular, reasons are computed by means of a data structure which maps literals to their corresponding reasons.

If during the propagation step an inconsistency is detected, DetConsH builds the reason of this inconsistency, stores it in $callerIR$ and returns false. Otherwise, an atom A is selected according to a heuristic criterion and MGBJ is recursively called on both $I \cup \{A\}$ and $I \cup \{\text{not } A\}$. If MGBJ returns true, a model has been found, otherwise an inconsistency has been determined. In the latter case, the reason of this inconsistency is stored in the variables $posIR$ or $negIR$, respectively. In both cases, when MGBJ fails, bj_level is compared with $curr\_level$. If bj_level is less than $current\_level$ we backjump to level bj_level, and return false. Otherwise, if an inconsistency has been found both for the positive and the negative assumption (both MGBJ subcalls failed) we perform the union of the two inconsistency reasons, discarding levels which are greater than or equal to current level (this is accomplished by the function Union)). The obtained set represents the reason of the failure of the sub-trees below the current assumption, and is recursively stored by using the $callerIR$ variable.

At this point, the next recursion level is computed by setting bj_level to the maximum of the set Union(posIR,negIR) or -1 if such a maximum does not exist (i.e., if

```
bool MGBJ (Interpretation& I, InconsistencyReason& callerIR, int& bj_level ) {
    bj_level ++;
    int curr_level = bj_level;
    if ( ! DetConsH ( I, curr_level, callerIR )
        return false;
    if ( “no atom is undefined in I” )
        return IsAnswerSet ( I );
    InconsistencyReason posIR, negIR;
    Select an undefined atom $A$ using a heuristic and set $R_A$=curr_level;
    if ( MGBJ( $I \cup \{A\}$, posIR, bj_level )
        return true;
    else
        callerIR = posIR;
        if (bj_level < curr_level)
            return false;
        else
            bj_level = curr_level;
            if ( MGBJ ( $I \cup \{\mathrm{not}\ A\}$, negIR, bj_level )
                return true;
            else
                callerIR = Union ( posIR, negIR );
                if ( bj_level < curr_level )
                    return false;
                bj_level = MAX ( callerIR );
                return false; };
```

Fig. 3. Computation of Answer Sets with backjumping

the set is empty). This corresponds to backtracking or backjumping to the closest recursion level involved in the conflict or, if the set Union(posIR,negIR) is empty, to abort the search because the problem is unsatisfiable. Such a situation occurs if the conflict depends only on the current choice, and we find an inconsistency whatever truth value we assume for it.

5 Comparison and Benchmarks

In order to evaluate the backjumping technique described in previous sections, we have implemented it as an experimental extension of the DLV system and we compared it to standard DLV by using 3-SAT problem instances. Conducting further experiments is on our agenda, but the picture and conclusions are rather clear when looking at just these experiments. We do not expect other benchmarks to be substantially different.

5.1 Experimental Results

Our experiments have been performed on a Power Mac G4 933 MHz machine with 1MB of Level 2 Cache and 256MB of RAM, running Mac OSX 10.2.8.

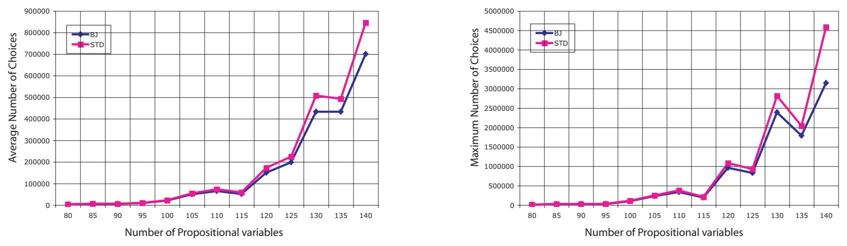

Fig. 4. Choice points on Random 3-SAT problems

We have generated 20 random 3-SAT instances for each problem size as indicated in [23]. The number of clauses for each generated instance was always 4.3 times the number of propositional variables (in order to generate hard instances).

We measured both the time required to solve each instance and the number of choices made by DLV. Time measurements have been done using the *time* command shipped with Mac OSX 10.2.8, counting total CPU time for the respective processes.

In Figure 4 we show the result obtained measuring choice points, while in Figure 5 we show time measurements. For every instance, we allowed a maximum running time of 3600 seconds (one hour). In each graph, the horizontal axis reports a parameter representing the size (number of variables) of the instance. In both cases, we disabled the lookahead heuristic, which is the default setting in DLV. In Figure 4, on the vertical axis, we report, respectively, the average and the maximum number of choices made over the 20 instances of the same size we run.

On average, DLV with backjumping (labelled BJ) requires a smaller number of choices than standard DLV (labelled STD) to solve an instance of the same size. The difference between BJ and STD becomes rather evident starting from instances of size 120, where BJ has made 150176 choices while STD has made 172282 choices (22106 more choices) on average, and the gap grows with the instance size. In fact, for instances of size 140, BJ has made 699792 choices, while STD has made 845649 choices (145857 more) on average. This trend is confirmed looking at the second graph in Figure 4, where BJ did at most 4582079 choices for an instance size of 140, which is more than one million choices less than STD, which required at most 3144722 choices.

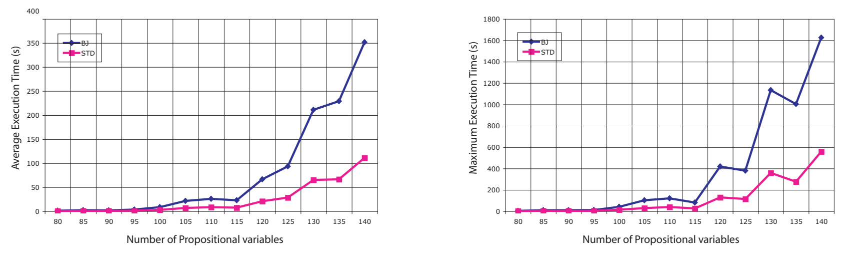

Fig. 5. Running Times on Random 3-SAT problems

Summarizing, we observed that the number of choices decreases of a non-negligible factor by using our backjumping technique.

The situation looks quite different when we look at Figure 5, where on the vertical axis, we report, respectively, the average and the maximum execution time required to solve the 20 instances of the same size. It is evident that STD outperforms BJ in these experiments. On average, STD took 110.8 seconds to solve an instance of 140 variables, while BJ took 351.53 seconds. This can be interpreted in the following way: On the average, BJ explores a smaller search tree, but the corresponding gain is compensated by the overhead introduced by the reason calculus (at least in our implementation).

We performed the same experiments with the lookahead heuristic enabled but we noticed virtually no difference in choice points between BJ and STD. This means that when lookahead is employed, there is basically no advantage when employing backjumping in addition – on the contrary, the overhead incurred by the reason calculus increases the computation time even more.

The ineffectiveness with respect to lookahead does not come as a big surprise. Since with lookahead (without the optimizations reported in [23]) DetCons will be activated for each atom upon assuming its truth and falsity, respectively, many inconsistent subtrees will be identified immediately after the relevant choice, in this way anticipating a large amount of potential backjumps.

6 Conclusion and Future Work

We have presented a backjumping technique for computing the answer sets of disjunctive logic programs. It is based on a reason calculus and is an elaboration of the work in [1, 2]. In particular, we do not build an implication graph, and thanks to the reason calculus determining the point to jump back to can be calculated more efficiently. Moreover, our framework is suitable for disjunctive programs.

We have implemented the technique in the DLV system, and have conducted several experiments with it. For these tests, backjumping without clause learning is not effective, unless one can come up with a highly optimized implementation of the reason calculus. In any case, backjumping without clause learning but with lookahead is clearly ineffective. We conjecture that backjumping needs clause learning in order to have a beneficial effect. However, our benchmark consists of randomly generated instances with little structure. It is possible that backjumping without clause learning is more effective on structured problems; we leave this issue for future studies.

Future work in this direction is therefore along two lines: 1. Refining the reason calculus and optimizing its implementation, and 2. implementing clause learning in DLV. The latter task is not trivial, as the DLV model generator currently heavily relies on the assumption that the program it works on is fixed. There are several ways of overcoming this difficulty, ranging from a redesign of the datastructures to an additional datastructure which is dedicated to the learned clauses. Finally, we plan to extend our experimentation in order to get a more complete picture of the impact of our new techniques. As pointed out by one of the reviewers, the reason calculus could also be exploited for debugging purposes.

References

1. Ward, J., Schlipf, J.S.: Answer Set Programming with Clause Learning. In: LPNMR-7. LNCS, (2004) 302–313
2. Ward, J.: Answer Set Programming with Clause Learning. PhD thesis, Ohio State University, Cincinnati, Ohio, USA (2004)
3. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM TODS **22** (1997) 364–418
4. Rintanen, J.: Improvements to the Evaluation of Quantified Boolean Formulae. In Dean, T., ed.: IJCAI 1999, Sweden,(1999) 1192–1197
5. Eiter, T., Gottlob, G.: The Complexity of Logic-Based Abduction. JACM **42** (1995) 3–42
6. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. CUP (2002)
7. Leone, N., Rosati, R., Scarcello, F.: Enhancing Answer Set Planning. In: IJCAI-01 Workshop on Planning under Uncertainty and Incomplete Information. (2001) 33–42
8. Janhunen, T., Niemelä, I., Simons, P., You, J.H.: Partiality and Disjunctions in Stable Model Semantics. In: KR 2000, 12-15,(2000) 411–419
9. Koch, C., Leone, N., Pfeifer, G.: Enhancing Disjunctive Logic Programming Systems by SAT Checkers. Artificial Intelligence **15** (2003) 177–212
10. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Declarative Problem-Solving Using the DLV System. In Minker, J., ed.: Logic-Based Artificial Intelligence. Kluwer (2000) 79–103
11. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC **9** (1991) 365–385
12. Przymusinski, T.C.: Stable Semantics for Disjunctive Programs. NGC **9** (1991) 401–424
13. Marek, W., Subrahmanian, V.: The Relationship between Logic Program Semantics and Non-Monotonic Reasoning. In: ICLP'89, MIT Press (1989) 600–617
14. Leone, N., Rullo, P., Scarcello, F.: Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation. Information and Computation **135** (1997) 69–112
15. Baral, C., Gelfond, M.: Logic Programming and Knowledge Representation. JLP **19/20** (1994) 73–148
16. Van Gelder, A., Ross, K., Schlipf, J.: The Well-Founded Semantics for General Logic Programs. JACM **38** (1991) 620–650
17. Ben-Eliyahu, R., Dechter, R.: Propositional Semantics for Disjunctive Logic Programs. AMAI **12** (1994) 53–87
18. Faber, W.: Enhancing Efficiency and Expressiveness in Answer Set Programming Systems. PhD thesis, TU Wien (2002)
19. Niemelä, I., Simons, P.: Efficient Implementation of the Well-founded and Stable Model Semantics. In Maher, M.J., ed.: ICLP'96, Bonn, Germany, MIT Press (1996) 289–303
20. Simons, P.: Extending and Implementing the Stable Model Semantics. PhD thesis, Helsinki University of Technology, Finland (2000)
21. Faber, W., Leone, N., Mateis, C., Pfeifer, G.: Using Database Optimization Techniques for Nonmonotonic Reasoning. In DDLP'99, Prolog Association of Japan (1999) 135–139
22. Faber, W., Leone, N., Pfeifer, G.: Experimenting with Heuristics for Answer Set Programming. In: IJCAI 2001, Seattle, WA, USA,(2001) 635–640
23. Faber, W., Leone, N., Pfeifer, G.: Optimizing the Computation of Heuristics for Answer Set Programming Systems. In: LPNMR'01. LNCS 2173
24. Faber, W., Leone, N., Pfeifer, G.: Pushing Goal Derivation in DLP Computations. In: LP-NMR'99. LNCS 1730
25. Calimeri, F., Faber, W., Leone, N., Pfeifer, G.: Pruning Operators for Answer Set Programming Systems. In: NMR'2002. (2002) 200–209