Look-Back Techniques and Heuristics in DLV: Implementation, Evaluation, and Comparison to QBF Solvers [⋆]

Marco Maratea [a,b] Francesco Ricca [a,*]
Wolfgang Faber [a] Nicola Leone [a]

[a] *Department of Mathematics, University of Calabria, 87036 Rende (CS), Italy*
[b] *DIST, University of Genova, 16145 Genova, Italy*

Abstract

DLV is the state-of-the-art system for evaluating disjunctive answer set programs. As in most Answer Set Programming (ASP) systems, its implementation is divided in a grounding part and a propositional model-finding part. In this paper, we focus on the latter, which relies on an algorithm using backtracking search.

Recently, DLV has been enhanced with backjumping techniques, which also involve a reason calculus, recording causes for the truth or falsity of atoms during the search. This reason calculus allows for looking back in the search process for identifying areas in the search space in which no answer set will be found. We can also define heuristics which make use of the information about reasons, preferring literals that were the reasons of more inconsistent branches of the search tree. This heuristics thus use information gathered earlier in the computation, and are therefore referred to as look-back heuristics.

In this paper, we formulate suitable look-back heuristics and focus on the experimental evaluation of the look-back techniques that we have implemented in DLV, obtaining the system DLV^{LB}. We have conducted a thorough experimental analysis considering both randomly-generated and structured instances of the 2QBF problem, the canonical problem for the complexity classes Σ_2^P and Π_2^P. Any problem in these classes can be expressed uniformly using ASP and can therefore be solved by DLV. We have also evaluated the same benchmark using "native" QBF solvers, which were among the best solvers in recent QBF Evaluations. The comparison shows that DLV endowed with look-back techniques is competitive with the best available QBF solvers on such instances.

Key words: Knowledge Representation and Reasoning, Nonmonotonic Reasoning, Heuristics

1 Introduction

Answer Set Programming (ASP) [1,2] is a purely declarative programming paradigm based on nonmonotonic reasoning and logic programming. The idea of answer set programming is to represent a given computational problem by a logic program the answer sets of which correspond to solutions, and then use an answer set solver to find such solutions [3]. The language of ASP is based on rules, allowing for both disjunction in rule heads and nonmonotonic negation in the body. ASP is very expressive, allowing for representing every property in the second level of the polynomial hierarchy. Therefore, ASP is strictly more expressive than using encodings based on satisfiability of propositional formulas (unless $P = NP$).

DLV is the state-of-the-art *disjunctive* ASP system, and it is based on an algorithm relying on backtracking search, like most other competitive ASP systems, which include the disjunctive solvers GnT [4] and Cmodels [5]. Recently, DLV has been enhanced with a backjumping procedure [6]. Backjumping [7,8] refers to an optimized recovery upon inconsistency during the search: instead of restoring the state of the search to the previous choice point, irrelevant choices for the encountered inconsistency are "jumped over", thereby restoring the search state to the previous *relevant* choice point. A crucial point is how relevance to an inconsistency can be determined. In [6], the necessary information for deciding relevance is recorded by means of a reason calculus, which collects information about the literals ("reasons") whose truth has caused the truth of other derived literals. Look-back heuristics [9] further strengthen the potential of backjumping by using the information made explicit by the reasons. The idea of this family of heuristics is to preferably choose those atoms which frequently caused inconsistencies. This significantly differs from classical ASP heuristics that use information arising from tentatively applying the simplification part (look-ahead) of the algorithm and analyzing the result. Look-back optimization techniques and heuristics have been shown, in various research areas, to be very effective on data intensive benchmarks coming from applications, like planning and formal verification (see, e.g., the reports of the various Competitions).

In this paper, we report on the formulation, implementation, and especially the experimental evaluation of look-back heuristics for DLV, yielding the system DLV^{LB}. Since the hardest problems that can be uniformly represented by

⋆ Preliminary versions of this work have been published at RCRA'07 and LPNMR'07.

* Corresponding author.

Email addresses: marco@dist.unige.it (Marco Maratea), ricca@mat.unical.it (Francesco Ricca), wf@wfaber.com (Wolfgang Faber), leone@mat.unical.it (Nicola Leone).

disjunctive logic programs, the language accepted by DLV^{LB}, are hard for the class Σ_2^P or Π_2^P, we have used the canonical problem for these classes, 2QBF—quantified boolean formulas with two alternating quantifiers, for evaluation purposes. In the literature of SAT a dichotomy has been reported, according to which random problem instances generally do not gain much from look-back techniques, while structured problem instances do - we have considered both types of problems in our experiments in order to assess whether a similar behavior can be observed for ASP.

DLV^{LB} provides several options regarding the initialization of the heuristics and the truth value to be assigned to an atom chosen by the heuristics. In our experimental analysis, we provide a comprehensive comparison of the impact of these options, and demonstrate how the new components of DLV^{LB} enhance the efficiency of DLV. We also provide a comparison to the other competitive disjunctive ASP systems GnT and Cmodels. Moreover, since we consider 2QBFs as a benchmark, we have also compared DLV^{LB} to the performance of native QBF solvers. In particular, we have chosen those solvers which were the best in recent QBF Evaluations over the various categories and which are freely available. As a result, we observe that DLV^{LB} clearly outperforms its direct competitors GnT and Cmodels, and that DLV^{LB} is also on par with the best available QBF solvers on 2QBF instances. Considering its knowledge representation merits and its computational competitiveness, we conjecture that DLV^{LB} is currently the system of choice for representing and solving problems which are on the second level of the polynomial hierarchy.

The paper is organized as follows: in Section 2 we review syntax, semantics and some properties of Answer Set Programming. In Section 3 we present first in Section 3.1 the basic algorithm underlying the DLV system and its extension by a reason calculus and backjumping. Since a main parameter of this algorithm is a heuristic choice to be made, in Section 3.2 we review some choice criteria from literature and define new look-back criteria in several variants. In Section 4 we then present the settings of the performed experiments, report the obtained results and discuss them. We discuss related work in Section 5 and draw our conclusions in Section 6.

2 Answer Set Programming Language

A *(disjunctive) rule* r is a formula

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

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are function-free atoms, $n \geq 0$, $m \geq k \geq 0$ and "not" is the nonmonotonic negation as failure operator. The disjunction $a_1 \lor$

$\cdots \vee a_n$ is the *head* of r, while $b_1, \cdots, b_k$, $\texttt{not}\ \ b_{k+1}, \cdots, \texttt{not}\ \ b_m$ is the *body*, of which $b_1, \cdots, b_k$ is the *positive body*, and $\texttt{not}\ \ b_{k+1}, \cdots, \texttt{not}\ \ b_m$ is the *negative body* of r. We will also denote the head and (positive or negative) body as sets containing the respective literals.

A rule r with empty negative body is called *positive*. A rule with empty head is referred to as *integrity constraint* or just *constraint*. If the body of a rule is empty we usually omit the :– sign.

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*. A rule is *safe* if each variable in that rule also appears in at least one positive literal in the body of that rule. A program is safe if each of its rules is safe and in the following we will assume that all programs are safe.

Example 1 *Consider the following ASP program* $\mathcal{P}_1$*:*

$$x(U)\ \vee\ x(V)\ :\!-\ d(U,V).$$
$$g :\!-\ x(U), x(V), d(U,V), \texttt{not}\ \ e(U,V).$$
$$:\!-\ g, x(U), x(V), d(U,V).$$
$$d(1,2).\quad d(3,4).\quad d(5,6).$$
$$e(1,5).\quad e(1,6).$$

The first rule is a positive and disjunctive rule, where its head is $\{x(U), x(V)\}$*, its positive body is* $\{d(U,V)\}$*, and its negative body is empty. The second rule is a non-disjunctive rule with head* $\{g\}$*, positive body* $\{x(U), x(V), d(U,V)\}$*, and negative body* $\{\texttt{not}\ \ e(U,V)\}$*. The third rule is a positive integrity constraint with empty head, positive body* $\{g, x(U), x(V), d(U,V)\}$*, and empty negative body. The last five rules are all ground facts. All of these rules are safe.*

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* $Ground(\mathcal{P})$ of $\mathcal{P}$ is the set $\bigcup_{r\in\mathcal{P}} Ground(r)$.

Example 2 *Reconsider program* $\mathcal{P}_1$ *of Example 1.* $U_{\mathcal{P}_1} = \{1, \ldots, 6\}$ *and* $B_{\mathcal{P}_1} = \{x(1), \ldots, x(6)\} \cup \bigcup_{(x,y)\in\{1,\ldots,6\}\times\{1,\ldots,6\}} \{e(x,y), d(x,y)\}$*. The program*

Ground$(\mathcal{P}_1)$ *is*

$$x(1) \lor x(1) \text{ :- } d(1,1).$$
$$x(1) \lor x(2) \text{ :- } d(1,2).$$
$$\vdots$$
$$x(6) \lor x(5) \text{ :- } d(6,5).$$
$$x(6) \lor x(6) \text{ :- } d(6,6).$$

$$g \text{ :- } x(1), x(1), d(1,1), \texttt{not}\ \ e(1,1).$$
$$g \text{ :- } x(1), x(2), d(1,2), \texttt{not}\ \ e(1,2).$$
$$\vdots$$
$$g \text{ :- } x(6), x(5), d(6,5), \texttt{not}\ \ e(6,5).$$
$$g \text{ :- } x(6), x(6), d(6,6), \texttt{not}\ \ e(6,6).$$

$$\text{:- } g, x(1), x(1), d(1,1), \texttt{not}\ \ e(1,1).$$
$$\text{:- } g, x(1), x(2), d(1,2), \texttt{not}\ \ e(1,2).$$
$$\vdots$$
$$\text{:- } g, x(6), x(5), d(6,5), \texttt{not}\ \ e(6,5).$$
$$\text{:- } g, x(6), x(6), d(6,6), \texttt{not}\ \ e(6,6).$$

$$d(1,2). \quad d(3,4). \quad d(5,6).$$
$$e(1,5). \quad e(1,6).$$

Note that the last five rules were already ground in $\mathcal{P}_1$*.*

For every program $\mathcal{P}$, its answer sets are defined using its ground instantiation *Ground*$(\mathcal{P})$ in two steps: first answer sets of positive programs are defined, then a reduction of general programs to positive ones is given, which is used 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 $\texttt{not}\ \ell$ is not contained in L. An interpretation I for $\mathcal{P}$ is a consistent set of ground literals over atoms in $B_{\mathcal{P}}$.[1] 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. Interpretation I is *total* if, for each atom A in

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

$B_{\mathcal{P}}$, either A or $\texttt{not}\ A$ 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 $r \in Ground(\mathcal{P})$, 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 it is minimal w.r.t. set inclusion among the models of $\mathcal{P}$.

Example 3 *For the positive ground program* $\mathcal{P}_2 = \{a \vee b \vee c.\,,\ \ {:\!-}\,a.\}$, *the two interpretations* $\{b, \texttt{not}\ a, \texttt{not}\ c\}$ *and* $\{c, \texttt{not}\ a, \texttt{not}\ b\}$ *are the only answer sets. For the positive ground program* $\mathcal{P}_3 = \{a \vee b \vee c.\,,\ \ {:\!-}\,a.\,,\ b\,{:\!-}\,c.\,,\ c\,{:\!-}\,b.\}$, $\{b, c, \texttt{not}\ a\}$ *is the only answer set.*

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 all rules $r \in \mathcal{P}$ the negative body of which 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 $Ground(\mathcal{P})^X$.

Example 4 *For the negative ground program* $\mathcal{P}_4 = \{a\,{:\!-}\,\texttt{not}\ b.\}$, $A = \{a, \texttt{not}\ \ b\}$ *is the only answer set, as* $\mathcal{P}_4^A = \{a.\}$. *For example for* $B = \{\texttt{not}\ \ a, b\}$, $\mathcal{P}_4^B = \emptyset$, *and so* B *is not an answer set.*

Example 5 *Reconsider program* $\mathcal{P}_1$ *from Example 1 and its grounding* $Ground(\mathcal{P}_1)$ *of Example 2. First, we note that any reduct of* $Ground(\mathcal{P}_1)$ *contains the following program* $\mathcal{P}_f$

$$
\begin{array}{l}
x(1)\ \vee\ x(1)\ {:\!-}\ d(1,1).\\
x(1)\ \vee\ x(2)\ {:\!-}\ d(1,2).\\
\vdots\\
x(6)\ \vee\ x(5)\ {:\!-}\ d(6,5).\\
x(6)\ \vee\ x(6)\ {:\!-}\ d(6,6).\\
{:\!-}\ g, x(1), x(1), d(1,1), \texttt{not}\ \ e(1,1).\\
{:\!-}\ g, x(1), x(2), d(1,2), \texttt{not}\ \ e(1,2).\\
\vdots\\
{:\!-}\ g, x(6), x(5), d(6,5), \texttt{not}\ \ e(6,5).\\
{:\!-}\ g, x(6), x(6), d(6,6), \texttt{not}\ \ e(6,6).\\
d(1,2).\quad d(3,4).\quad d(5,6).\\
e(1,5).\quad e(1,6).
\end{array}
$$

So any minimal model of a reduct, and thus any answer set, must contain $F^+ = \{d(1,2), d(3,4), d(5,6), e(1,5), e(1,6)\}$. *It is also easy to see that any minimal model of a reduct must contain* $F^- = \bigcup_{(x,y)\in\{1,\ldots,6\}\times\{1,\ldots,6\}}\{\texttt{not}\ \ d(x,y), \texttt{not}\ \ e(x,y)\}\setminus\{\texttt{not}\ \ d(1,2), \texttt{not}\ \ d(3,4), \texttt{not}\ \ d(5,6), \texttt{not}\ \ e(1,5), \texttt{not}\ \ e(1,6)\}$ *as there is no reason for the truth of the atoms in* F^-. *So that means that for any possible answer set* $A \supset F^+ \cup F^-$, *the reduct* $Ground(\mathcal{P}_1)^A$ *contains* $\mathcal{P}_f$ *and*

$$
\begin{aligned}
&g \text{ :- } x(1), x(1), d(1,1).\\
&g \text{ :- } x(1), x(2), d(1,2).\\
&g \text{ :- } x(1), x(2), d(1,3).\\
&g \text{ :- } x(1), x(2), d(1,4).\\
&g \text{ :- } x(1), x(1), d(2,1).\\
&g \text{ :- } x(1), x(2), d(2,2).\\
&\vdots\\
&g \text{ :- } x(6), x(5), d(6,5).\\
&g \text{ :- } x(6), x(6), d(6,6).
\end{aligned}
$$

The reducts of all possible answer sets are therefore equal, and so it is sufficient to isolate those $A \supset F^+ \cup F^-$ *which are minimal models of this reduct program, and thus answer sets:*

$$
\begin{aligned}
&\{x(1), \texttt{not}\ \ x(2), x(3), \texttt{not}\ \ x(4), x(5), \texttt{not}\ \ x(6)\} \cup F^+ \cup F^-\\
&\{\texttt{not}\ \ x(1), x(2), x(3), \texttt{not}\ \ x(4), x(5), \texttt{not}\ \ x(6)\} \cup F^+ \cup F^-\\
&\{x(1), \texttt{not}\ \ x(2), \texttt{not}\ \ x(3), x(4), x(5), \texttt{not}\ \ x(6)\} \cup F^+ \cup F^-\\
&\{\texttt{not}\ \ x(1), x(2), \texttt{not}\ \ x(3), x(4), x(5), \texttt{not}\ \ x(6)\} \cup F^+ \cup F^-\\
&\{x(1), \texttt{not}\ \ x(2), x(3), \texttt{not}\ \ x(4), \texttt{not}\ \ x(5), x(6)\} \cup F^+ \cup F^-\\
&\{\texttt{not}\ \ x(1), x(2), x(3), \texttt{not}\ \ x(4), \texttt{not}\ \ x(5), x(6)\} \cup F^+ \cup F^-\\
&\{x(1), \texttt{not}\ \ x(2), \texttt{not}\ \ x(3), x(4), \texttt{not}\ \ x(5), x(6)\} \cup F^+ \cup F^-\\
&\{\texttt{not}\ \ x(1), x(2), \texttt{not}\ \ x(3), x(4), \texttt{not}\ \ x(5), x(6)\} \cup F^+ \cup F^-
\end{aligned}
$$

3 Answer Set Computation Algorithms

In this section, we briefly describe the main steps of the computational process performed by ASP systems. We will refer particularly to the computational

```
bool ModelGenerator ( Interpretation& I ) {
    I = DetCons ( I );
    if ( I == $\mathcal{L}$ ) then
       return false;
    if ( “no atom is undefined in I” )
       return IsAnswerSet(I);
    Select an undefined atom $A$ using a heuristic;
    if ( ModelGenerator ( $I \cup \{A\}$ ) )
       return true;
    else
       return ModelGenerator ( $I \cup \{\texttt{not}\ A\}$ );
};
```

Fig. 1. Computation of Answer Sets without backjumping.

engine of the DLV system, which will be used for the experiments, but also other ASP systems employ a similar procedure. In general, an answer set program $\mathcal{P}$ contains variables. The first step of a computation of an ASP system eliminates these variables, generating a ground instantiation $ground(\mathcal{P})$ of $\mathcal{P}$.[2] The subsequent computations, which constitute the non-deterministic core of the system, are then performed on $ground(\mathcal{P})$ by the so called Model Generator procedure.

In the following paragraphs, we briefly illustrate the original model generation algorithm of DLV and an enhancement of it by means of a backjumping technique. Finally, we report a description of all the heuristics, that will later be compared in the experiments.

3.1 The Model Generator Algorithms

Note that the algorithms presented here are abstractions of actual implementations, which have to deal with several additional technical details and optimizations. For more details we refer to [11] for the basic technique and to [6] for the enhancement by backjumping. Moreover, the algorithms presented here compute one answer set for simplicity, however they can be modified to compute all or n answer sets in a straightforward way.

The basic method is the Model Generator Algorithm sketched in Figure 1.This function is initially called with parameter I set to the empty interpretation, in which all atoms are undefined.[3]

[2] Note that $ground(\mathcal{P})$ is usually not the full $Ground(\mathcal{P})$; rather, it is a subset (often much smaller) of it having precisely the same answer sets as $\mathcal{P}$ [10].
[3] Observe that the interpretations built during the computation are 3-valued, that is, a literal can be True, False or Undefined w.r.t. I.

If the program $\mathcal{P}$ has an answer set, then the function returns True, setting I to the computed answer set; otherwise it returns False. The Model Generator is similar to the DPLL procedure employed by SAT solvers. It first calls a function DetCons, which returns the extension of I with the literals that can be deterministically inferred (or the set of all literals $\mathcal{L}$ upon inconsistency). This function is similar to a unit propagation procedure employed by SAT solvers, but exploits the peculiarities of ASP for making further inferences (e.g., it exploits 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 called on $I \cup \{A\}$ and on $I \cup \{\texttt{not}\ A\}$. The atom A plays the role of a branching variable of a SAT solver. And indeed, like for SAT solvers, the selection of a "good" atom A is crucial for the performance of an ASP system. In Section 3.2, we will describe some heuristic criteria for the selection of such branching atoms.

If no atom is left for branching, the Model Generator has produced a "candidate" answer set, the stability of which is subsequently verified by *IsAnswerSet(I)*. This function checks whether the given "candidate" I is a minimal model of the program $Ground(\mathcal{P})^I$ and if so, outputs I. *IsAnswerSet(I)* returns True if the computation should be stopped and False otherwise. Note that, if during the execution of the ModelGenerator function a contradiction arises, or the stable model candidate is not a minimal model, ModelGenerator backtracks and modifies the last choice. This kind of backtracking is called chronological backtracking.

To give an intuition on how backjumping is supposed to work, consider the following simple example.

Consider the following program, which is a simplified ground version of $\mathcal{P}_1$ of Example 1.

$$\begin{array}{llllll}
r_1: & x(1) \vee x(2). & r_2: & x(3) \vee x(4). & r_3: & x(5) \vee x(6). \\
r_4: & g \text{:-} x(1), x(5). & r_5: & \text{:-}\, g, x(1), x(5). & & \\
r_6: & g \text{:-} x(1), x(6). & r_7: & \text{:-}\, g, x(1), x(6). & &
\end{array}$$

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

Here we first assume $x(1)$ to be true, deriving $x(2)$ to be false (from r_1 to ensure the minimality of answer sets). Then we assume $x(3)$ to be true, deriving $x(4)$ to be false (from r_2 for minimality). Third, we assume $x(5)$ to be true and derive $x(6)$ to be false (from r_3 for minimality) and g to be true (from r_4 by forward inference). This truth assignment violates constraint r_5 (where g must be false), yielding an inconsistency. We continue the search by inverting the last choice, that is, we assume $x(5)$ to be false and we derive $x(6)$ to be true

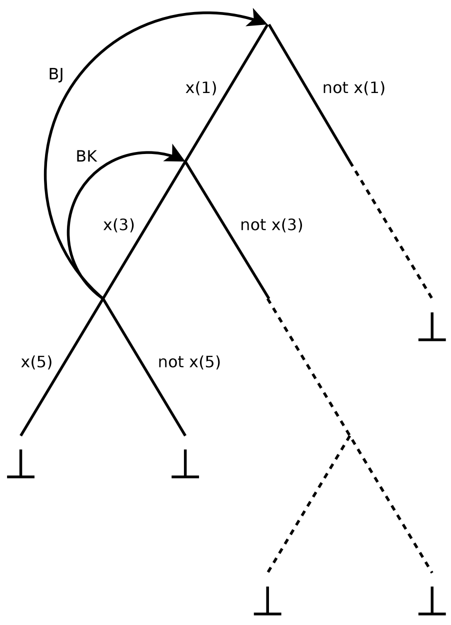

Fig. 2. Backtracking vs backjumping.

(again from r_3 to preserve minimality) and g to be true (from r_6 by forward inference), but obtain another inconsistency (because constraint r_7 is violated, here g must also be false).

At this point, ModelGenerator goes back to the previous choice point, in this case inverting the truth value of $x(3)$ (cf. the arc labelled BK in Fig. 2).

Now it is important to note that the inconsistencies obtained are independent of the choice of $x(3)$, and only the truth value of $x(1)$ and $x(5)$ are the reasons for the encountered inconsistencies. In fact, no matter what the truth value of $x(3)$ is, if $x(1)$ is true then any truth assignment for $x(5)$ will lead to an inconsistency. Looking at Fig. 2, this means that in the whole subtree below the arc labelled $x(1)$ no stable model can be found. It is therefore obvious that the chronological backtracking search explores branches of the search tree that cannot contain a stable model, performing a lot of useless work. A better policy would be to go back directly to the point at which we assumed $x(1)$ 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.

In practice, once a literal has been assigned a truth value during the computation, we can associate a reason for that fact with the literal. For instance, given a rule a :− $b, c,$ `not` $d.$, if b and c are true and d is false in the current partial interpretation, then a will be derived as true (by Forward Propagation). In this case, we can say that a is true "because" b and c are true and d is false. A special case are *chosen* literals, as their only reason is the fact that they have been chosen. The chosen literals can therefore be seen as being their own reason, and we may refer to them as elementary reasons. All other reasons are consequences of elementary reasons, and hence aggregations of elementary reasons. Each literal l derived during the propagation (i.e., DetCons) will have an associated set of positive integers $R(l)$ representing the reason of l, which are essentially the recursion levels of the chosen literals which entail l. Therefore, for any chosen literal c, $|R(c)| = 1$ holds.

The process of defining reasons for derived (non-chosen) literals is called *reason calculus*. For a detailed definition of this calculus we refer to [6].

When an inconsistency is determined, we use reason information in order to understand which chosen literals have to be undone in order to avoid the found inconsistency. Implicitly this also means that all choices which are not in the reason do not have any influence on the inconsistency. We can isolate two main types of inconsistencies: (*i*) deriving conflicting literals, and (*ii*) failing stability checks. Of these two, the second one is a peculiarity of disjunctive ASP.

Deriving conflicting literals means, in our setting, that DetCons determines that an atom a and its negation `not` a should both hold. In this case, the reason of the inconsistency is – rather straightforward – the combination of the reasons for a and `not` a: $R(a) \cup R(\texttt{not}\ a)$.

Inconsistencies from failing stability checks are a peculiarity of disjunctive ASP, as non-disjunctive ASP systems usually do not employ a stability check. This situation occurs if the function IsAnswerSet(I) of ModelGenerator returns false, hence if the checked interpretation (which is guaranteed to be a model) is not stable. The reason for such an inconsistency is always based on an unfounded set, which has been determined inside IsAnswerSet(I) as a side-effect. Using this unfounded set, the reason for the inconsistency is composed of the reasons of literals which satisfy rules containing unfounded atoms in their head (the cancelling assignments of these rules). The information on reasons for inconsistencies can be exploited for backjumping by going back to the closest choice which is a reason for the inconsistency, rather than always to the immediately preceding choice.

The function ModelGeneratorBJ (shown in Fig. 3) is a modification of the ModelGenerator function, which implements backjumping. To this end, two new parameters IR and bj_level are introduced, which hold the reason of the inconsistency encountered in the subtrees whose current recursion level is the root, and the recursion level to backtrack or backjump to. When going forward in recursion, bj_level is also used to hold the current level. The variables $curr\_level$, $posIR$, and $negIR$ are local to ModelGeneratorBJ and used for holding the current recursion level, and the reasons for the positive and negative recursive branch, respectively.

```
bool ModelGeneratorBJ (Interpretation& I, Reason& IR,
                        int& bj_level ) {

    bj_level ++;
    int curr_level = bj_level;

    I = DetConsBJ ( I, IR );
    if ( I == $\mathcal{L}$ ) return false;
    if ( “no atom is undefined in I” )
      if IsAnswerSetBJ( I, IR ); return true;
      else
        bj_level = MAX ( IR );
        return false;

    Reason posIR, negIR;

    Select an undefined atom $A$ using a heuristic;

    $R(A)$= { curr_level };
    if ( ModelGeneratorBJ( $I \cup \{A\}$, posIR, bj_level )
      return true;
    if (bj_level < curr_level)
      IR = posIR; return false;

    bj_level = curr_level;
    $R(\texttt{not}\ A)$ = { curr_level };
    if ( ModelGeneratorBJ ( $I \cup \{\texttt{not}\ A\}$, negIR, bj_level )
      return true;

    if ( bj_level < curr_level )
      IR = negIR; return false;

    IR = trim( curr_level, Union ( posIR, negIR ) );
    bj_level = MAX ( IR );
    return false;
};
```

Fig. 3. Computation of Answer Sets with backjumping.

Instead of DetCons, here DetConsBJ is used, which additionally computes the reasons of the inferred literals and if it encounters an inconsistency it will return the reason of this inconsistency in its second parameter IR. Instead of IsAnswerSet, ModelGeneratorBJ uses IsAnswerSetBJ, which additionally computes the inconsistency reason in case of a failure of the stability check, returning it in its second parameter.

Whenever there is the possibility to backjump, we set *bj_level* to the maximal level of the inconsistency reason (or 0 if it is the empty set) before returning from this instance of ModelGeneratorBJ, the idea being that the maximum level in IR corresponds to the nearest (chronologically) choice causing the failure.

The information provided by reasons can be further exploited in a backjumping-based solver. In particular, in the following paragraph we describe how reasons for inconsistencies can be exploited for defining look-back heuristics.

3.2 Heuristics

In this paragraph we will first describe the two main heuristics for DLV (based on look-ahead), and subsequently define several new heuristics based on reasons (or based on look-back), which are computed as side-effects of the backjumping technique. We assume that a ground ASP program $\mathcal{P}$ and an interpretation I have been fixed. We first recall the "standard" DLV heuristic h_{UT} [12], which has recently been refined to yield the heuristic h_{DS} [13], which is more "specialized" for hard disjunctive programs (like 2QBF). These are look-ahead heuristics, that is, the heuristic value of a literal Q depends on the result of taking Q true and computing its consequences. Given a literal Q, $ext(Q)$ will denote the interpretation resulting from the application of DetCons on $I \cup \{Q\}$; w.l.o.g., we assume that $ext(Q)$ is consistent, otherwise Q is automatically set to false and the heuristic is not evaluated on Q at all.

Standard Heuristic of DLV (h_{UT}). This heuristic, which is still the default in the DLV distribution, has been proposed in [12], where it was shown to be very effective on many relevant problems. It exploits a peculiar property of ASP, namely *supportedness*: for each true atom A of an answer set I, there exists a rule r of the program such that the body of r is true w.r.t. I and A is the only true atom in the head of r. Since an ASP system must eventually converge to a supported interpretation, h_{UT} is geared towards choosing those literals which minimize the number of *UnsupportedTrue (UT)* atoms, i.e., atoms which are true in the current interpretation but still miss a supporting rule. The heuristic h_{UT} is "balanced", that is, the heuristic values of an atom Q depend on both the effect of taking Q and `not` Q, the decision between Q and `not` Q is based on the UT atoms criteria.

Enhanced Heuristic of DLV (h_{DS}). The heuristic h_{DS} [14] is based on h_{UT}, and is different from h_{UT} only for pairs of literals which are not ordered by h_{UT}. The idea of the additional criterion is that interpretations having a "higher degree of supportedness" are preferred, where the degree of supportedness is the average number of supporting rules for the true atoms. Intuitively, if all true atoms have many supporting rules in a model M, then the elimination of a true atom from the interpretation would violate many rules, and it becomes less likely finding a subset of M which is a model of $\mathcal{P}^M$ (which would disprove that M is an answer set). Interpretations with a higher degree of supportedness are therefore more likely to be answer sets. Just like h_{UT}, h_{DS} is "balanced".

The Family of Look-back Heuristics (h_{LB}). We next describe a family of new look-back heuristics h_{LB}, motivated by heuristics implemented in SAT solvers like Chaff [9]. Different to h_{UT} and h_{DS}, which provide a partial order on potential choices, h_{LB} assigns a number $V(L)$ to each literal L (thereby inducing an implicit order). The basic idea is that this number is periodically updated using information about the inconsistencies that the assumption of the respective literal caused. The effect of basing this value on previous chosen literals is twofold: first, it causes literals to be chosen that have also previously been chosen in a different subtree of the search, thereby avoiding blindly choosing literals of which nothing is known yet. Second, it favors the choice of literals which are more likely to lead to inconsistent sub-branches, which in general has the effect of more likely generating a smaller search tree[4]. Whenever a literal is to be selected, the literal with the largest $V(L)$ will be chosen. If several literals have the same $V(L)$, then negative literals are preferred over positive ones, but among negative and positive literals having the same $V(L)$, the ordering will be random.

In more detail, for each literal L, two values are stored: $V(L)$, the current heuristic value, and $I(L)$, the number of inconsistencies L has been a reason for since the most recent heuristic value update. After having chosen k literals, $V(L)$ is updated for each L as follows: $V(L) := V(L)/2 + I(L)$. The division is often referred to as "aging" and has the effect of giving more importance to recent data. The division itself is assumed to be defined on integers by rounding the result. It is important to note that $I(L) \neq 0$ can hold only for literals that have been chosen earlier (in a temporal sense, thus in a previously explored branch of the search tree) during the computation. This criterion is fairly simple and obviously very efficient to compute once the reason calculus is used. Especially compared to the previously described look-ahead based criteria, which involve a fairly heavy computation for almost every literal, computing this criterion takes negligible time, especially if backjumping is employed, in which the reasons have to be calculated and maintained anyway.

A crucial point left basically unspecified by the definition so far are the initial values of $V(L)$. Given that, initially, no information about inconsistencies is available, all $V(L)$ will initially be 0, and so a random choice would be taken. It is not immediately clear how to define this initialization in the best way. Yet, initializing these values seems to be crucial, as making poor choices in

[4] Note that the search tree is not stored in its entirety in DLV, but only one partial branch at a time.

the beginning of the computation can be fatal for efficiency, especially since the heuristics favor choosing literals that have already been chosen earlier.

Here, we present two possibilities for initialization: the first, denoted by h_{LB}^{MF}, is done by initializing $V(L)$ to the number of occurrences of L in the program rules. The motivation for h_{LB}^{MF} is that it is fast to compute and stays with the "no look-ahead" and "most constrained first" ideas of h_{LB}. The second initialization, denoted by h_{LB}^{LF}, involves ordering the atoms with respect to h_{DS}, and initializing $V(L)$ by the rank in this ordering. The motivation for h_{LB}^{LF} is to try to use as much initialization as possible initially, as the first choices can be critical for the size of the subsequent computation tree, as h_{LB} implicitly prefers choosing atoms that have already been taken.

We also introduce yet another option for h_{LB}, motivated by the fact that answer sets for disjunctive programs must be minimal with respect to atoms interpreted as true, and the fact that the checks for minimality are costly: if false literals are chosen preferably, then the computed answer set candidates may have a better chance to be already minimal. The heuristics h_{LB} already prefers false literals having the same $V(L)$ as positive ones, but we can go one step further and completely ignore the polarity of the best literals with respect to $V(L)$, choosing always the negative literal containing the atom of the best literal, even if it is positive and the corresponding negative literal has a lower value. This really means to consider, for each atom A, the value $max(V(A), V(\texttt{not}\ \ A))$, and then choose the negative literal containing the best atom according to that value. If we employ this option in the heuristics, we denote it by adding AF to the superscript, arriving at $h_{LB}^{MF,AF}$ and $h_{LB}^{LF,AF}$ respectively.

4 Experiments

We have implemented the above-mentioned look-back techniques and heuristics in DLV; in this section, we report on their experimental evaluation.

4.1 Compared Methods

For our experiments, we have compared several versions of DLV [15], which differ on the employed heuristics and the use of backjumping. For having a broader picture, we have also compared our implementations to the competing systems GnT and CModels3, and with the QBF solvers ssolve and sKizzo. The considered systems are:

• **dlv.ut**, the standard DLV system employing h_{UT} (based on look-ahead).
• **dlv.ds**, DLV with h_{DS}, the look-ahead based heuristic specialized for Σ_2^P/Π_2^P hard disjunctive programs.
• **dlv.ds.bj**, DLV with h_{DS} and backjumping.
• **dlv.mf.bj**, DLV with h_{LB}^{MF} and backjumping.
• **dlv.mf.af.bj**: DLV with $h_{LB}^{MF,AF}$ and backjumping.
• **dlv.lf.bj**, DLV with h_{LB}^{LF} and backjumping.
• **dlv.lf.af.bj**, DLV with $h_{LB}^{LF,AF}$ and backjumping.
• **gnt** [4]: the solver GnT, based on the Smodels system, can deal with disjunctive ASP. One instance of Smodels generates candidate models, while another instance tests if a candidate model is stable.
• **cm3** [5]: CModels3, a solver based on the definition of completion for disjunctive programs and the extension of loop formulas to the disjunctive case. CModels3 uses two SAT solvers in an interleaved way, the first for finding answer set candidates using the completion of the input program and loop formulas obtained during the computation, the second for verifying if the candidate model is indeed an answer set. In the experiments, we used zChaff (ver. 2004) as underlying SAT solver: it is the default and fastest SAT solver among the ones available in CModels3.
• **ssolve** [16]: is a search based native QBF solver that won the QBF Evaluation in 2004 on random (or probabilistic) benchmarks (performing very well also on non-random, or fixed, benchmarks), and performed globally (i.e., both on fixed and probabilistic benchmarks) well in the last two editions.
• **sKizzo** [17]: is a reasoning engine for QBF featuring several techniques, including search, resolution and skolemization, that won the last QBF Evaluation 2007 (which was run only on fixed benchmarks).
• **quantor** [18]: is a QBF solver based on Q-resolution (to eliminate existential variables) and Shannon expansion (to eliminate universal variables), plus a number of features, such as equivalence reasoning, subsumption checking, pure literal detection, unit propagation, and also a scheduler for the elimination step.

For h_{LB} heuristics we fixed k=100. We have conducted further experiments with different values for k which indicate that 100 is a local optimum. Since the basic picture does not seem to change significantly with different values for k, we do not report on these experiments here. Note that we have not taken into account other solvers like Smodels$_{cc}$ [19] or Clasp [20] because our focus is on disjunctive ASP.

Furthermore, we want to point out that the QBF solvers which we have evaluated, besides being among the best in recent QBF Evaluations, also represent the three main lines of research for implementing QBF solvers: (*i*) search-based extension of the DLL algorithm for SAT (ssolve), (*ii*) quantifier elimination and Q-resolution (quantor), and (*iii*) hybrid method (sKizzo). Note also that the performance comparison to QBF solvers is not to be seen as a competition, as these systems are fairly different in nature: on the one hand, QBF solvers can deal with arbitrary QBFs, not just 2QBFs, thus also PSPACE-hard problems. On the other hand, ASP can make use of variables, and in this way allows for uniformly expressing all problems on the second level of the polynomial hierarchy. The goal of our analysis is therefore to check whether the runtimes of DLV are acceptable in comparison.

4.2 Benchmark Programs and Data

The proposed heuristic aims at improving the performance of DLV on disjunctive ASP programs. Therefore we focus on hard programs in this class, which is known to be able to express each query of the complexity class Σ_2^P/Π_2^P, and can therefore be considered to be the canonical problem for these complexity classes. All of the instances that we have considered in our benchmark analysis have been derived from instances for 2QBF, the canonical problem for the second level of the polynomial hierarchy. This choice is motivated by the fact that many real-world, structured (i.e., fixed) instances in this complexity class are available for 2QBF on QBFLIB [21,22], and moreover, studies on the location of hard instances for randomly generated 2QBFs have been reported in [23–25].

The problem 2QBF consists of deciding whether a quantified Boolean formula (QBF) $\Phi = \forall X \exists Y \phi$, where X and Y are disjoint sets of propositional variables and $\phi = D_1 \wedge \ldots \wedge D_k$ is a CNF formula over $X \cup Y$, is valid.

The transformation from 2QBF to disjunctive logic programming is a slightly altered form of a reduction used in [26]. The propositional disjunctive logic program $\mathcal{P}_\phi$ produced by the transformation requires $2*(|X|+|Y|)+1$ propositional predicates (with one dedicated predicate w), and consists of the following rules:

$v \vee \bar{v}.$	for each variable $v \in X \cup Y$
$y \leftarrow w.\ \bar{y} \leftarrow w.$	for each variable $y \in Y$
$w \leftarrow v_{m+1}, \ldots, v_n, \bar{v}_1, \ldots, \bar{v}_m.$	for each disjunction $\neg v_{m+1} \vee \cdots \vee \neg v_n \vee$ $\vee\, v_1 \vee \cdots \vee v_m$ in ϕ
$\leftarrow \texttt{not}\ w.$	

The 2QBF formula Φ is valid iff $\mathcal{P}_\Phi$ has no answer set [26].

Example 6 *The 2QBF* $\Phi = \forall x \exists y[(\neg x \vee y) \wedge (\neg y \vee x)]$ *is transformed into*

$\mathcal{P}_\Phi = \{x \vee \bar{x}.;\ y \vee \bar{y}.;\ y \leftarrow w.;\ \bar{y} \leftarrow w.;\ w \leftarrow x, \bar{y}.;\ w \leftarrow y, \bar{x}.;\ \leftarrow \texttt{not}\ w.\}$.

$\mathcal{P}_\Phi$ does not have an answer set, thus the 2QBF ϕ is valid. To check this manually, observe that Φ is equivalent to $\forall x \exists y : x \leftrightarrow y$, and indeed for each valuation for x we can find a valuation for y (namely the same as for x) such that the equivalence holds.

We have selected both random and structured 2QBF instances. The random 2QBF instances have been generated following recent phase transition results for QBFs [23–25]. In particular, the generation method described in [25] has been employed and the generation parameters have been chosen according to the experimental results reported in the same paper. First, we have generated 10 different sets of instances, each of which is labelled with an indication of the employed generation parameters. In particular, the label "A-E-C-ρ" indicates the class of instances in which each clause has A universally-quantified variables and E existentially-quantified variables randomly chosen from a set containing C variables, such that the ratio between universal and existential variables is ρ. For example, the instances in the class "3-3-70-0.8" are 6CNF formulas (each clause having exactly 3 universally-quantified variables and 3 existentially-quantified variables) whose variables are randomly chosen from a set of 70 containing 31 universal and 39 existential variables, respectively. In order to compare the performance of the systems in the vicinity of the phase transition, each set of generated formulas has an increasing ratio of clauses over existential variables (from 1 to max$r$). Following the results presented in [25], max$r$ has been set to 21 for each of the classes 3-3-70-*, and 12 for each of 2-3-80-*. We have generated 10 instances for each ratio, thus obtaining, in total, 210 and 120 instances, respectively. Then, because these instances do not provide information about the scalability of the systems w.r.t. the total number of variables, we generated yet more sets. We took the "2-3-80-1.0" and "3-3-70-1.2" classes, fixed the ratio of clauses over existential variables to the "harder" value for the DLV versions and vary the number of variables C (from 5 to max$C$, step 5), where max$C$ is 80 and 70, respectively. We have generated 10 instances for each point, thus obtaining, in total, 160 and 140 instances per set, respectively.

Concerning the structured instances, we have analyzed:

- **Narizzano-Robot** - These are real-word instances encoding the robot navigation problems presented in [27], as used in the QBF Evaluation 2004 and 2005.
- **Ayari-MutexP** - These 2QBFs encode instances to problems related to the formal equivalence checking of partial implementations of circuits, as presented in [28].
- **Letz-Tree** - These instances consist of simple variable-independent subprograms generated according to the pattern: $\forall x_1 x_3 ... x_{n-1} \exists x_2 x_4 ... x_n (c_1 \wedge \ldots \wedge$

	dlv.ut	dlv.ds	dlv.ds.bj	dlv.mf.bj	dlv.mf.af.bj	dlv.lf.bj	dlv.lf.af.bj	gnt	cm3	ssolve	sKizzo	quantor
2-3-80-0.4	119	**120**	**120**	**120**	**120**	**120**	**120**	3	**57**	**120**	38	41
2-3-80-0.6	91	**102**	99	**103**	83	101	96	4	**62**	**120**	25	32
2-3-80-0.8	88	**99**	**99**	**99**	79	97	92	5	**73**	**120**	21	29
2-3-80-1.0	81	95	**96**	**106**	80	95	95	10	**81**	**120**	21	26
2-3-80-1.2	84	99	**101**	**109**	85	101	102	6	**93**	**120**	22	27
3-3-70-0.6	159	**174**	168	**172**	157	164	166	4	**76**	**210**	49	66
3-3-70-0.8	128	**138**	135	**150**	123	132	140	2	**82**	**210**	37	53
3-3-70-1.0	114	**128**	127	**149**	112	128	125	7	**96**	**205**	34	50
3-3-70-1.2	123	131	**133**	**156**	115	129	140	9	**117**	**209**	34	47
3-3-70-1.4	124	139	**142**	**161**	117	142	141	9	**131**	**210**	34	43
#Total	1111	1225	1220	1325	1071	1209	1217	59	868	1644	315	414

Table 1
Number of solved instances within timeout for Random 2QBF.

$c_{n-2})$ where $c_i = x_i \vee x_{i+2} \vee x_{i+3}$, $c_{i+1} = \neg x_i \vee \neg x_{i+2} \vee \neg x_{i+3}$, $i = 1, 3, \ldots, n-3$.

The benchmark instances belonging to Letz-tree, Narizzano-Robot, Ayari-MutexP have been obtained from QBFLIB [21], including the 32 (resp. 40) Narizzano-Robot instances used in the QBF Evaluation 2004 (resp. 2005), and all the $\forall\exists$ instances from Letz-tree and Ayari-MutexP.

4.3 Results

All the experiments were performed on a 3GHz PentiumIV equipped with 1GB of RAM, 2MB of level 2 cache running Debian GNU/Linux. Time measurements have been done using the `time` command shipped with the system, counting total CPU time for the respective process.

We start with the results of the experiments with random 2QBF formulas. For every instance, we have allowed a maximum running time of 20 minutes. In Table 1 we report, for each system, the number of instances solved in each set within the time limit, highlighting the best value for each group of systems. Looking at the table, it is clear that the new look-back heuristic combined with the "mf" initialization (corresponding to the system dlv.mf.bj) performed very well on these domains, being the version which was able to solve most instances in most settings among the ASP systems, particularly on the 3-3-70-* sets. Also dlv.lf.bj, in particular when combined with the "af" option, performed quite well, while the other variants do no seem to be very effective. Considering the look-ahead versions of DLV, dlv.ds performed reasonably well. Considering GnT and CModels3, we note that they solve quite few instances, while it is clear that ssolve is very efficient, being able to solving almost all instances. In contrast, sKizzo and quantor did not perform well here, which is in line with the results of the QBF Evaluations which showed that ssolve is very efficient on probabilistic (i.e., fixed) benchmarks, while sKizzo and quantor are not efficient on this domain.

Figures 4 and 5 show the results for the "2-3-80-1.0", Figures 6 and 7 for the "3-3-70-1.2" set, regarding scalability. For the sake of readability, only the instances with a high number of variables are presented: GnT, Cmodels3, ssolve, sKizzo, quantor and all the DLV versions solve all instances not reported. Figures 4 and 6 contain the cumulative number of solved instances for all DLV versions while Figures 5 and 7 contain the respective data for GnT, CModels3, ssolve, sKizzo, quantor and the best version of DLV. Overall, on these particular sets, we can see that all the "look-back" versions of DLV scaled much better than GnT and CModels3, with dlv.mf.bj being able to solve some of the bigger instances not solved by other DLV versions, GnT and Cmodels3. ssolve managed to solve all instances (but one in Fig. 6), and in shorter time (not reported), while sKizzo and quantor showed comparatively poor performances.

In Tables 2, 3 and 4, we report the results, in terms of execution time for finding one answer set, and/or number of instances solved within 20 minutes, about the groups: Narizzano-Robot, Ayari-MutexP and Letz-Tree, respectively. The last columns (AS?) indicate whether the instance has an answer set (Y), or

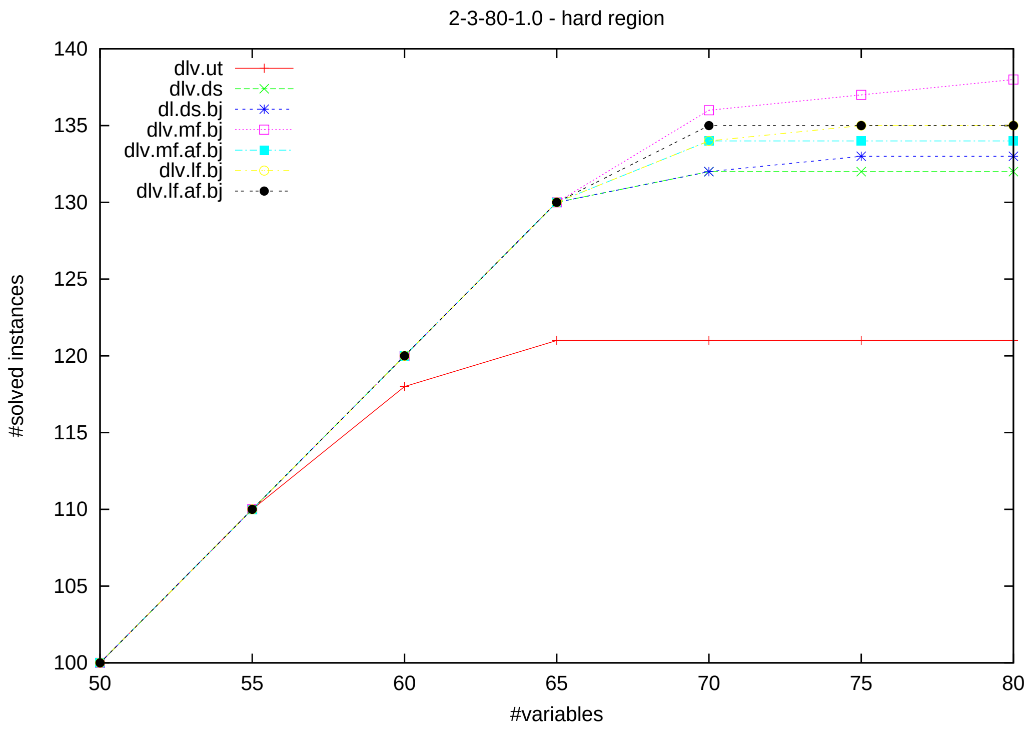

Fig. 4. Number of solved instances by all DLV versions.

not (N): only in Table 2 it indicates how many instances have answer sets. A "–" in these tables indicates a timeout or a memory out.

In Table 2 we report only the instances from the QBF Evaluation 2004 and 2005, respectively, which were solved within the time limit by at least one of the compared methods. In Table 2, dlv.mf.bj was, among the ASP and QBF solvers, the system which solved the highest number of instances among the 67 reported (24 for QBF Evaluation 2004 and 40 for QBF Evaluation 2005) instances, followed by ssolve (60), CModels3 and sKizzo (58), and dlv.lf.bj (50). Moreover, dlv.mf.bj solved a superset of the instances solved by ssolve, while the timeouts of dlv.mf.bj showed up on different instances w.r.t. the timeouts of sKizzo. Further, dlv.mf.bj was always the fastest ASP system on each instance (sometimes drastically, even if for the sake of presentation we do

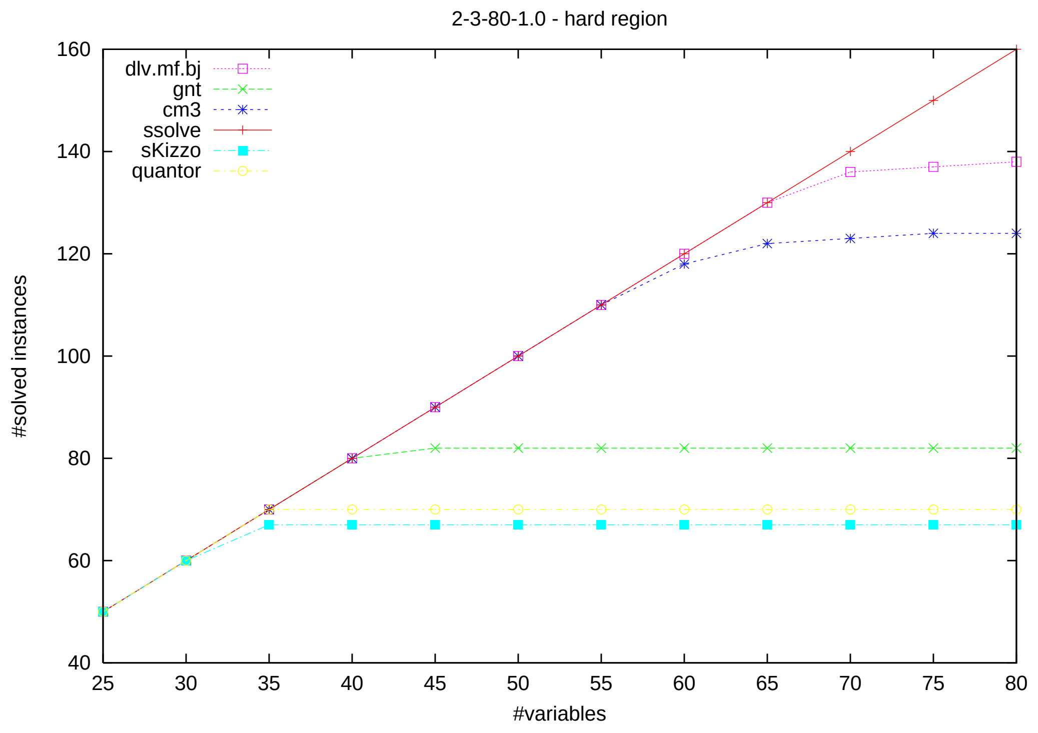

Fig. 5. Number of solved instances by dlv.mf.bj, GnT, CModels3, ssolve, sKizzo and quantor.

	dlv.ut	dlv.ds	dlv.ds.bj	dlv.mf.bj	dlv.mf.af.bj	dlv.lf.bj	dlv.lf.af.bj	gnt	cm3	ssolve	sKizzo	quantor	AS?
QBFEval.2004	10	10	11	24	15	18	13	5	18	20	22	10	5
QBFEval.2005	0	0	10	40	34	32	22	0	40	40	36	0	0
#Total	10	10	21	64	49	50	35	5	58	60	58	10	5

Table 2
Number of solved instances on Narizzano-Robot instances as selected in the QBF Evaluation 2004 and 2005.

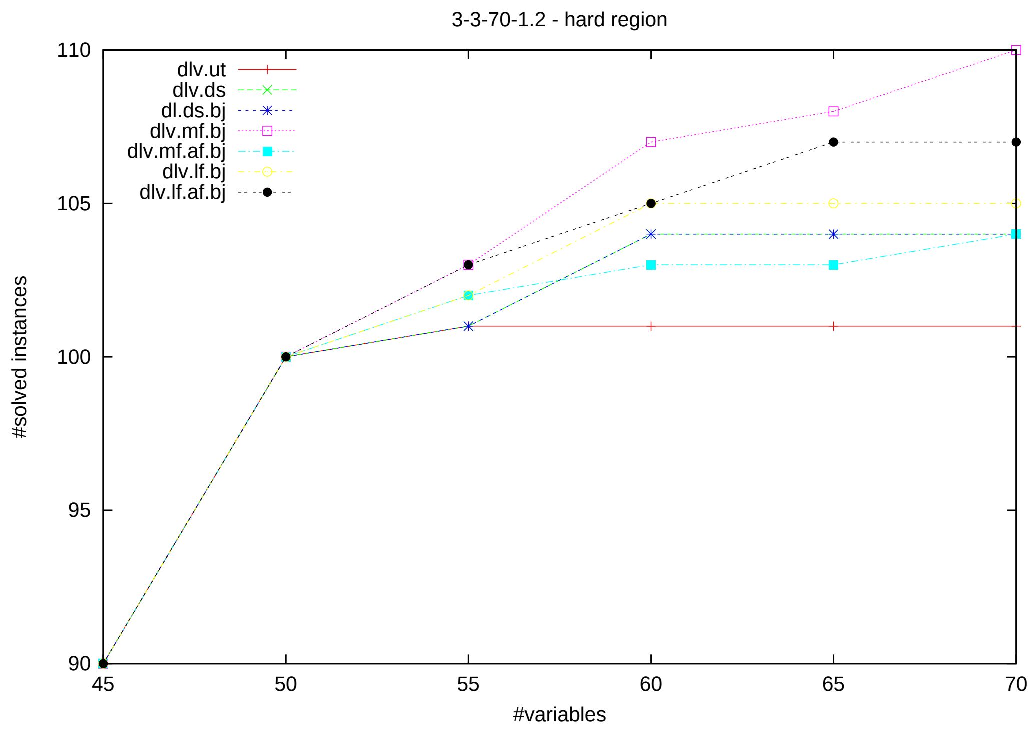

Fig. 6. Number of solved instances by all DLV versions.

not report CPU time) if we consider the instances on which it took more than 1 second, and often faster than ssolve and sKizzo, especially on the QBF Evaluation 2004 instances. All of the QBF Evaluation 2005 instances were solved by dlv.mf.bj, Cmodels3 and ssolve, with mean execution times of 228.07s, 189.74s and 76.91s, respectively. The "traditional" DLV versions could solve 10 instances, while dlv.ds.bj solved 21 instances, and took less execution time. This indicates the advantages of using a backjumping technique on these robot instances.

In Table 3, we then report the results for Ayari-MutexP. In that domain all the versions of DLV and the QBF solvers ssolve and quantor were able to solve all 7 instances, outperforming both CModels3 and GnT which solved only one instance. Comparing the execution times required by all the variants of DLV we note that, also in this case, dlv.mf.bj is the best-performing version, while QBF solvers scaled up much better, but for quantor that quickly run out of memory.

About the Letz-Tree domain reported in Table 4, the DLV versions equipped with look-back heuristics solved a higher number of instances and required less CPU time (up to two orders of magnitude less) than all ASP competitors. In particular, the look-ahead based versions of DLV, GnT and CModels3 could solve only 3 instances, while dlv.mf.bj and dlv.lf.bj solved 4 and 5 instances, respectively. Interestingly, here the "lf" variant is very effective, in particular

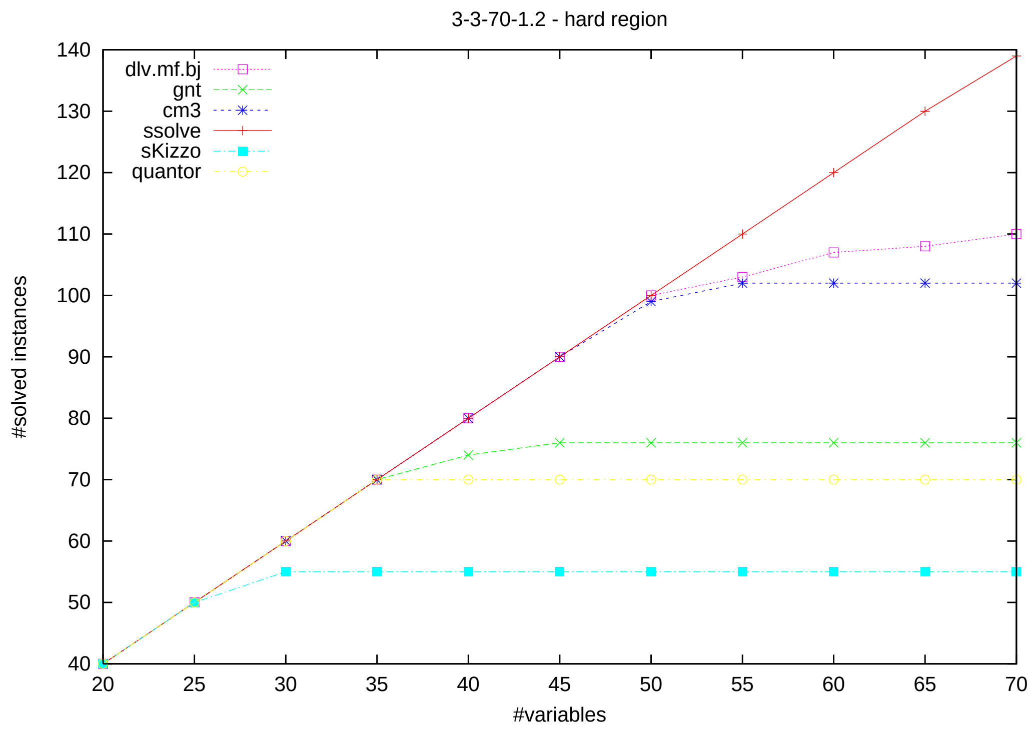

Fig. 7. Number of solved instances by dlv.mf.bj, GnT, CModels3, ssolve, sKizzo and quantor.

	dlv.ut	dlv.ds	dlv.ds.bj	dlv.mf.bj	dlv.mf.af.bj	dlv.lf.bj	dlv.lf.af.bj	gnt	cm3	ssolve	sKizzo	quantor	AS?
mutex-2-s	0.01	0.01	0.01	0.01	0.01	0.01	0.01	1.89	0.65	0.03	0.01	0.01	N
mutex-4-s	0.05	0.05	0.05	0.06	0.05	0.06	0.05	–	–	0.04	0.01	0.01	N
mutex-8-s	0.21	0.2	0.23	0.21	0.21	0.23	0.21	–	–	0.07	0.01	0.7	N
mutex-16-s	0.89	0.89	0.98	0.89	0.89	1.01	0.9	–	–	0.13	0.01	–	N
mutex-32-s	3.67	3.72	4.06	3.63	3.64	4.16	3.79	–	–	0.3	0.03	–	N
mutex-64-s	15.38	16.08	17.64	14.97	15.04	18.08	16.97	–	–	0.81	0.07	–	N
mutex-128-s	69.07	79.39	90.92	62.97	62.97	92.92	93.05	–	–	2.83	0.13	–	N
#Solved	7	7	7	7	7	7	7	1	1	7	7	3	

Table 3
Execution time (seconds) and number of solved instances on Ayari-MutexP instances.

when combined with the "af" option, like in the random instances for testing scalability. It could solve the same number of instances as ssolve, sKizzo and quantor, which, however, scale better.

4.4 Strategic companies

We also run native disjunctive ASP benchmarks for the *Strategic Companies* problem, as defined in [29]. The goal here is to understand how the new look-back based DLV versions perform on these instances. We have also transformed the ASP intput into QBFs for having a complete picture also in this case. A similar analysis in [13], however, showed that QBF solvers generally do not perform very well on this kind of input.

Here, we generated tests as in [15] with 20 instances each size for m companies ($5 \leq m \leq 100$), $3m$ products, 10 uniform randomly chosen *contr_by* relations per company (up to four controlling companies), and uniform randomly chosen *prod_by* relations (up to four producers per product), for a total of 400 instances. The problem is deciding whether two fixed companies (1 and 2, without loss of generality) are strategic.

For the QBF solvers we have produced the following formula:

$$\exists c_1, \ldots, c_n : \forall c'_1, \ldots, c'_n : ((I \wedge NE) \rightarrow (R \wedge R') \wedge c_1 \wedge c_2)$$

where I stands for

$$(c'_1 \rightarrow c_1) \wedge \ldots \wedge (c'_n \rightarrow c_n)$$

NE for

$$\neg((c'_1 \leftrightarrow c_1) \wedge \ldots \wedge (c'_n \leftrightarrow c_n))$$

R for

$$\bigwedge_{i=1}^{m}((\bigwedge_{c_j \in O_i} c_j) \rightarrow c_i) \wedge \bigwedge_{i=1}^{n}(\bigvee_{g_i \in C_j} c_j)$$

	dlv.ut	dlv.ds	dlv.ds.bj	dlv.mf.bj	dlv.mf.af.bj	dlv.lf.bj	dlv.lf.af.bj	gnt	cm3	ssolve	sKizzo	quantor	AS?
exa10-10	0.18	0.17	0.17	0.04	0.1	0.06	0.06	0.12	0.03	0.01	0.01	0.01	N
exa10-15	7.49	7.09	7.31	0.34	0.71	0.48	0.38	6.46	0.73	0.01	0.01	0.01	N
exa10-20	278.01	264.53	275.1	12.31	17.24	5.43	2.86	325.26	67.56	0.02	0.01	0.01	N
exa10-25	–	–	–	303.67	432.32	44.13	19.15	–	–	0.02	0.02	0.01	N
exa10-30	–	–	–	–	–	166.93	129.54	–	–	0.05	0.02	0.02	N
#Solved	3	3	3	4	4	5	5	3	3	5	5	5	

Table 4
Execution time (seconds) and number of solved instances on Letz-Tree instances.

	dlv.ut	dlv.ds	dlv.ds.bj	dlv.mf.bj	dlv.mf.af.bj	dlv.lf.bj	dlv.lf.af.bj	gnt	cm3	ssolve	sKizzo	quantor
#Solved	400	400	400	400	400	400	396	400	400	195	91	119

Table 5
Number of solved instances on Strategic Companies.

and O_i contains the controlling companies of c_i, while C_j contains the companies producing good j. R' is defined analogous to R on the primed variables.

Unfortunately this formula is not in CNF, as required by the qDimacs format. In order to avoid a substantial blowup of the formula by a trivial normalization, we have used the tool *qst* of the *traquasto* suite [30], which transforms a formula into qDimacs by introducing additional "label variables" to avoid exponential formula growth. However, these additional variables are existentially quantified at the inner level and thus would turn the formula above into a 3QBF. To avoid this, we consider the negated formula

$$\forall c_1, \ldots, c_n : \exists c'_1, \ldots, c'_n : \neg((I \wedge NE) \rightarrow (R \wedge R') \wedge c_1 \wedge c_2)$$

which stays on the second level after the transformation.

In Table 5 we report the total number of solved instances. We can see that all DLV versions (but dlv.lf.af.bj), GnT and Cmodels3 are able to solve all the generated instances, while ssolve, sKizzo and quantor can just solve a very limited portion, i.e., the smallest instances in the set.

Summarizing, in all of the test cases presented, both random and structured, DLV equipped with look-back heuristics obtained good results both in terms of number of solved instances and execution time compared to traditional DLV, GnT and CModels3. Variant dlv.mf.bj, the "classic" look-back heuristic, performed best in most cases, but good performance was obtained also by dlv.lf.bj. The results of dlv.lf.af.bj on the some random and Letz-Tree instances show that this option can be fruitfully exploited in some particular domains. The QBF solvers ssolve and sKizzo in general performed very well, but on some domains (notably Narizzano-Robot for both solvers, and also random benchmarks for sKizzo) they were outperformed by dlv.mf.bj, both in terms of number of instances solved and CPU execution time. On the other hand, quantor performed well on some domains, but in others, i.e., Narizzano-Robot and Ayari-Mutex, it could solve very few instances, often because it run out of memory. Moreover, ASP systems did much better than QBF solvers in the Strategic Companies benchmarks. Overall we can observe that look-back based ASP systems, in particular dlv.mf.bj, are competitive with QBF solvers. It should be also noted that the vast majority of the structured instances presented do not have answer sets, while the bigger advantages of dlv.mf.bj over ssolve on the Narizzano-Robot instances are obtained on the instances with answer sets.

5 Related Work

In this section we provide an overview about related work, especially with respect to backjumping and look-back heuristics, and outline the main differences to our approach.

Look-back techniques, including backjumping notions, first studied for constraint solving [7,8,31], have been applied successfully for SAT [32–34,9] and QBF solving [35–38]. More recently, they have been ported to ASP, resulting in the non-disjunctive systems Smodels$_{cc}$ [19,39] and Clasp [20], and ultimately also disjunctive ones, like CModels3 [5] and DLV [6]. In this paper, we have extended the latter work, which provided a backjumping-enabled DLV, by a variety of look-back heuristics, resulting in the system DLV^{LB}.

We will discuss the differences of our work with respect to related computational engines by considering the following groups of systems:

(1) CSP and SAT solvers
(2) non-disjunctive ASP systems
(3) disjunctive ASP systems
(4) QBF solvers

About group (1), in which the use of look-back techniques originated, we observe that the formalism for determining reasons for inconsistencies is quite different from the one presented in [6], and thus to the one in DLV^{LB}.

First of all, we note that DLV^{LB} makes use of reasons for concepts that do not have any counterpart in formalisms and systems of group (1), like the notion of unfounded sets, and stability check failures. Moreover, unlike CSP and SAT solvers, DLV^{LB} does not use an implication graph, which has a similar role as the reason table of DLV^{LB}. In practice, in both CSP and SAT solvers, the implication relationships of variable assignments made during the computation is stored in a directed graph (the implication graph), which contains a node *for each* variable assignment. Two nodes, say a and b, are connected by an arc from a to b whenever b is implied by a during a propagation step. A crucial difference between the implication graph and the reason table is that the first one stores also the dependencies between "implied" assignments, while in the latter only branching literals are taken into account. Moreover, reasons of conflicts are computed in a different way in DLV^{LB} and systems in group (1). In particular, most SAT solvers build that reason from a clause which corresponds to a "vertex cut" [9] in the implication graph. This cut partitions the implication graph in two sets of nodes, the first containing all the branching variables (called *reason side*), and the latter containing the conflicting variable *conflict side*. In general, there are many possible “cuts”, and the reason computed in DLV^{LB} basically coincides with the particular one which contains only the branching assignments in the reason side. However, many SAT solvers employ the so-called 1UIP (First Unique Implication Point) cut [9], which, in general, is very different from the one corresponding to DLV^{LB} inconsistency reasons. [5]

As a consequence of all these differences, both reasons and heuristic values are quite different in DLV^{LB} with respect to the standard methods in SAT and CSP solvers. One can argue that the heuristics in those systems are somewhat more “informed”, since also implied atoms can occur in the reason of a conflict. This additional information is obtained by (i) maintaining the implication graph, which can be done efficiently [6], but in a more involved way than the reason table of DLV^{LB}, and (ii) exploiting a process (determining the 1UIP cut) quite more intricate than the one implemented in DLV^{LB}. Indeed, reasons are computed by performing the union of two sets of integers in DLV^{LB}, compared to a more entangled transversal of the implication graph.

It is worth noting that, for systems in group (1), look-back techniques are virtually always combined with other two techniques that DLV^{LB} does not employ. The first one, called clause-learning, heavily relies on the implication graph, and allows for pruning the search space by adding new clauses representing conflicts. The second one, which is always used in combination with clause learning, requires to periodically restart the search from the beginning by retaining the newly added clauses.

As the experiments clearly demonstrate, DLV equipped with backjumping and look-back heuristics behaves very well, even without learning and restart. However, we plan to add learning and restart capabilities to DLV in order to further enhance its performance. Doing so requires rather fundamental changes inside DLV, which currently heavily relies on the assumption that the ground program does not change during the computation.

Concerning the systems in group (2), such as Smodels$_{cc}$ [19] or Clasp [20], almost all arguments as for those in group (1) apply, apart from the fact that the concept of unfounded sets indeed exists and is considered in the reason calculus of those systems.

The only system (apart from DLV and DLV^{LB}) in group (3) that uses look-back techniques is CModels3[40,5]. However, this system relies on a SAT solver

[5] The 1UIP cut is preferred since it plays a central role in the clause learning technique, which is usually combined with backjumping in SAT solvers.

[6] Note that the implication graph can be implemented by associating each implied variable with a pointer to the clause from which it has been derived.

internally, and the look-back techniques are confined to the SAT subsystem, so the respective comments for SAT apply. Nevertheless, CModels3 tries to bias the SAT heuristics by adding clauses in the event of model check failures, which is loosely related to recording the reasons in case of a model check failure in DLV^{LB}.

Finally, about group (4), given the nature of the problem, two types of backjumping are applicable: conflict- and solution-directed backjumping, following the terminology of [37] (cumulatively called dependency-directed backtracking in [38]). The first type allows search to skip over existentially quantified literals while backtracking, while the second allows the same behavior on universally quantified literals. Both types of conflicts can be used in order to update the heuristic values. The conflict-directed backjumping is a direct extension of the method used in SAT, and thus most of the discussions for group (1) apply also in this case.

6 Conclusions

We have described a general framework for employing look-back techniques in disjunctive ASP, based on the reason calculus described in [6], which allows for the design of a variety of look-back based heuristics. In this work, we have defined a basic heuristics h_{LB}, which together with two different initialization strategies yields the heuristics h_{LB}^{MF} and h_{LB}^{LF}. In addition, we have also defined a criterion in which the negative literal is always chosen first, regardless of the polarity of the best literal according to the heuristics, arriving at variants $h_{LB}^{MF,AF}$ and $h_{LB}^{LF,AF}$. We have implemented all proposed techniques in the DLV system, and carried out a broad experimental analysis on hard instances encoding 2QBFs, comprising both randomly generated instances , generated according to the method proposed in [25], and structured instances from the QBFLIB archive.

It turned out that the proposed heuristics outperform the traditional (disjunctive) ASP systems DLV, GnT and CModels3 in most cases, and a rather simple approach (“dlv.mf.bj”) works particularly well, being performance-wise competitive with respect to “native” QBF solvers. A possible topic for future research is to further expand the range of look-back techniques in DLV by employing *learning* (the ability to record reasons in order to further avoid inconsistencies already encountered).

Acknowledgements

This work has been supported by M.I.U.R. within projects “Potenziamento e Applicazioni della Programmazione Logica Disgiuntiva” and “Sistemi basati sulla logica per la rappresentazione di conoscenza: estensioni e tecniche di ottimizzazione.” We would like to thank the anonymous reviewers for their useful suggestions.

References

[1] M. Gelfond, V. Lifschitz, The Stable Model Semantics for Logic Programming, in: Logic Programming: Proceedings Fifth Intl Conference and Symposium, MIT Press, Cambridge, Mass., 1988, pp. 1070–1080.

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

[3] V. Lifschitz, Answer Set Planning, in: D. D. Schreye (Ed.), Proceedings of the 16th International Conference on Logic Programming (ICLP'99), The MIT Press, Las Cruces, New Mexico, USA, 1999, pp. 23–37.

[4] T. Janhunen, I. Niemelä, Gnt - a solver for disjunctive logic programs, in: V. Lifschitz, I. Niemelä (Eds.), Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-7), Vol. 2923 of LNAI, Springer, 2004, pp. 331–335.

[5] Y. Lierler, Disjunctive Answer Set Programming via Satisfiability, in: C. Baral, G. Greco, N. Leone, G. Terracina (Eds.), Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR'05, Diamante, Italy, September 2005, Proceedings, Vol. 3662 of Lecture Notes in Computer Science, Springer Verlag, 2005, pp. 447–451.

[6] F. Ricca, W. Faber, N. Leone, A Backjumping Technique for Disjunctive Logic Programming, AI Communications – The European Journal on Artificial Intelligence 19 (2) (2006) 155–172.

[7] J. Gaschnig, Performance measurement and analysis of certain search algorithms, Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA, Technical Report CMU-CS-79-124 (1979).

[8] P. Prosser, Hybrid Algorithms for the Constraint Satisfaction Problem., Computational Intelligence 9 (1993) 268–299.

[9] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an Efficient SAT Solver, in: Proceedings of the 38th Design Automation Conference, DAC 2001, ACM, Las Vegas, NV, USA, 2001, pp. 530–535.

[10] W. Faber, N. Leone, C. Mateis, G. Pfeifer, Using Database Optimization Techniques for Nonmonotonic Reasoning, in: INAP Organizing Committee (Ed.), Proceedings of the 7th International Workshop on Deductive Databases and Logic Programming (DDLP'99), Prolog Association of Japan, 1999, pp. 135–139.

[11] W. Faber, Enhancing Efficiency and Expressiveness in Answer Set Programming Systems, Ph.D. thesis, Institut für Informationssysteme, Technische Universität Wien (2002).

[12] W. Faber, N. Leone, G. Pfeifer, Experimenting with Heuristics for Answer Set Programming, in: Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI) 2001, Morgan Kaufmann Publishers, Seattle, WA, USA, 2001, pp. 635–640.

[13] W. Faber, F. Ricca, Solving Hard ASP Programs Efficiently, in: C. Baral, G. Greco, N. Leone, G. Terracina (Eds.), Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR'05, Diamante, Italy, September 2005, Proceedings, Vol. 3662 of Lecture Notes in Computer Science, Springer Verlag, 2005, pp. 240–252.

[14] W. Faber, N. Leone, F. Ricca, Solving Hard Problems for the Second Level of the Polynomial Hierarchy: Heuristics and Benchmarks, Intelligenza Artificiale 2 (3) (2005) 21–28.

[15] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, F. Scarcello, The DLV System for Knowledge Representation and Reasoning, ACM Transactions on Computational Logic 7 (3) (2006) 499–562.

[16] R. Feldmann, B. Monien, S. Schamberger, A Distributed Algorithm to Evaluate Quantified Boolean Formulae, in: Proceedings National Conference on AI (AAAI'00), AAAI Press, Austin, Texas, 2000, pp. 285–290.

[17] M. Benedetti, skizzo: A suite to evaluate and certify qbfs., in: R. Nieuwenhuis (Ed.), Automated Deduction - CADE-20, 20th International Conference on Automated Deduction, Tallinn, Estonia, Vol. 3632 of Lecture Notes in Computer Science, Springer, 2005, pp. 369–376.

[18] A. Biere, Resolve and Expand., in: Revised Selected Papers of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04), Vol. 3542 of Lecture Notes in AI (LNAI), 2005, pp. 59–70.

[19] J. Ward, J. S. Schlipf, Answer Set Programming with Clause Learning, in: V. Lifschitz, I. Niemelä (Eds.), Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-7), Vol. 2923 of LNAI, Springer, 2004, pp. 302–313.

[20] M. Gebser, B. Kaufmann, A. Neumann, T. Schaub, Conflict-driven answer set solving, in: Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07), Morgan Kaufmann Publishers, 2007, pp. 386–392.

[21] M. Narizzano, A. Tacchella, QBF Solvers Evaluation page, http://www.qbflib.org/qbfeval/index.html/ (2002).

[22] M. Narizzano, L. Pulina, A. Tacchella, The QBFEVAL Web Portal, in: Proceedings of the 10th European Conference on Logics in Artificial Intelligence (JELIA'06), Vol. 4160 of Lecture Notes in AI (LNAI), Springer, 2006, pp. 494–497.

[23] M. Cadoli, A. Giovanardi, M. Schaerf, Experimental Analysis of the Computational Cost of Evaluating Quantified Boolean Formulae., in: Proceedings of the 5th Congress: Advances in Artificial Intelligence of the Italian Association for Artificial Intelligence, AI*IA 97, Lecture Notes in Computer Science, Springer Verlag, Rome, Italy, 1997, pp. 207–218.

[24] I. Gent, T. Walsh, The QSAT Phase Transition, in: Proceedings of the 16th AAAI, 1999.

[25] H. Chen, Y. Interian, A model for generating random quantified boolean formulas., in: Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05), Professional Book Center, 2005, pp. 66–71.

[26] T. Eiter, G. Gottlob, On the Computational Cost of Disjunctive Logic Programming: Propositional Case, Annals of Mathematics and Artificial Intelligence 15 (3/4) (1995) 289–323.

[27] C. Castellini, E. Giunchiglia, A. Tacchella, SAT-based planning in complex domains: Concurrency, constraints and nondeterminism., Artificial Intelligence 147 (1/2) (2003) 85–117.

[28] A. Ayari, D. A. Basin, Bounded Model Construction for Monadic Second-Order Logics., in: Prooceedings of Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, 2000.

[29] M. Cadoli, T. Eiter, G. Gottlob, Default Logic as a Query Language, IEEE Transactions on Knowledge and Data Engineering 9 (3) (1997) 448–463.

[30] M. Zolda, Comparing Different Prenexing Strategies for Quantified Boolean Formulas, Master's thesis, Institut für Informationssysteme, Technische Universität Wien (2005).

[31] R. Dechter, D. Frost, Backjump-based backtracking for constraint satisfaction problems., Artificial Intelligence 136 (2) (2002) 147–188.

[32] R. Bayardo, R. Schrag, Using CSP Look-back Techniques to Solve Real-world SAT Instances, in: Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-97), 1997, pp. 203–208.

[33] J. P. M. Silva, K. A. Sakallah, GRASP: A Search Algorithm for Propositional Satisfiability, IEEE Transaction on Computers 48 (5) (1999) 506–521.

[34] E. Goldberg, Y. Novikov, BerkMin: A Fast and Robust Sat-Solver, in: Design, Automation and Test in Europe Conference and Exposition (DATE 2002), IEEE Computer Society, Paris, France, 2002, pp. 142–149.

[35] L. Zhang, S. Malik, Conflict Driven Learning in a Quantified Boolean Satisfiability Solver, in: Proceedings of the International Conference on Computer-Aided Design (ICCAD 2002), 2002, pp. 442–449.

[36] L. Zhang, S. Malik, Towards a Symmetric Treatment of Satisfaction and Conflicts in Quantified Boolean Formula Evaluation., in: Proceedings of Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Lecture Notes in Computer Science, Springer Verlag, Ithaca, NY, USA, 2002, pp. 200–215.

[37] E. Giunchiglia, M. Narizzano, A. Tacchella, Backjumping for Quantified Boolean Logic Satisfiability, Artificial Intelligence 145 (2003) 99–120.

[38] R. Letz, Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas, in: Proceedings of Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX 2002, Lecture Notes in Computer Science, Springer Verlag, Copenhagen, Denmark, 2002, pp. 160–175.

[39] J. Ward, Answer Set Programming with Clause Learning, Ph.D. thesis, Ohio State University, Cincinnati, Ohio, USA (2004).

[40] E. Giunchiglia, Y. Lierler, M. Maratea, Answer Set Programming Based on Propositional Satisfiability, Journal of Automated Reasoning 36 (4) (2006) 345–377.