A Backjumping Technique for Disjunctive Logic Programming

Francesco Ricca [*], Wolfgang Faber and Nicola Leone

Department of Mathematics, University of Calabria.

87030 Rende (CS), Italy

E-mail: {ricca, faber, leone}@mat.unical.it

In this work we present a backjumping technique for Disjunctive Logic Programming under the Stable Model Semantics (SDLP). It builds upon related techniques that had originally been introduced for constraint solving, which have been adapted to propositional satisfiability testing, and recently also to non-disjunctive logic programming under the stable model semantics (SLP) [1,2].

We focus on backjumping without clause learning, providing a new theoretical framework for backjumping in SDLP, elaborating on and exploiting peculiarities of the disjunctive setting. We present a reason calculus and associated computations, which – compared to the traditional approaches – reduces the information to be stored, while fully preserving the correctness and the efficiency of the backjumping technique, handling specific aspects of disjunction in a benign way. We implemented the proposed technique in DLV, the state-of-the-art SDLP system.

We have conducted several experiments on hard random and structured instances in order to assess the impact of backjumping. To this end, we have compared DLV in various versions: With and without the backjumping method described in this paper, in combination with two different heuristic functions. Our conclusion is that under any of the heuristic functions, DLV with backjumping is favourable to DLV without backjumping. DLV with backjumping performs particularly well on structured satisfiability and quantified boolean formula instances, where the search space and execution time are effectively cut.

Keywords: Logic Programming, Stable Models, Backjumping, Experiments

[*]Corresponding author.

1. Introduction

SDLP. Disjunctive Logic Programming under the Stable Model Semantics (SDLP) [1], is a novel programming paradigm, which has been proposed in the area of nonmonotonic reasoning and logic programming. The idea of SDLP is to represent a given computational problem by a logic program whose stable models correspond to solutions, and then use a solver to find such a solution [3].

The knowledge representation language of SDLP is very expressive in a precise mathematical sense; in its general form, allowing for disjunction in rule heads and nonmonotonic negation in rule bodies, SDLP can represent *every* problem in the complexity class Σ_2^P and Π_2^P (under brave and cautious reasoning, respectively) [4]. Thus, SDLP is strictly more powerful than SAT-based programming (unless some widely believed complexity assumptions do not hold), as it allows us to solve even problems which cannot be translated to SAT in polynomial time. The high expressive power of SDLP can be profitably exploited in AI, which often has to deal with problems of high complexity. For instance, problems in diagnosis and planning under incomplete knowledge are complete for the the complexity class Σ_2^P or Π_2^P [5,6], and can be naturally encoded in SDLP [7,8].

As an example, consider the well-known 3COLORABILITY problem. Given a graph $G = (V, E)$, assign each node one of three colors (say, red, green, or blue) such that adjacent nodes always have distinct colors. The SDLP encoding is as follows:

$$vertex(v). \qquad \forall v \in V$$

$$edge(v_1, v_2). \qquad \forall (v_1, v_2) \in E$$

$$col(X, red) \vee col(X, green) \vee col(X, blue) \;\texttt{:-}\; vertex(X).$$

$$\texttt{:-}\; edge(X, Y), col(X, C), col(Y, C).$$

[1]Often SDLP is referred to as Answer Set Programming (ASP). While ASP supports also a second (“strong”) kind of negation, it can be simulated in SDLP. To avoid confusion, we will only use the term SDLP in this paper.

AI Communications
ISSN 0921-7126, IOS Press. All rights reserved

The first two lines introduce suitable facts, representing G, the third line states that each vertex needs to have some color. Since stable models are minimal w.r.t. set inclusion, each vertex will be associated to precisely one color in any stable model. The last line acts as an integrity constraint and disallows situations in which two vertices connected by an edge are associated with the same color. By now, several systems are available, which implement SDLP: DLV [9], GnT [10], and recently cmodels-3 [11].

Main Issues. Most of the optimization work on related SDLP systems has focused on the efficient evaluation of non-disjunctive programs (whose power is limited to NP/co-NP), whereas the optimization of full SDLP has been treated in fewer works (e.g., in [10,12]).

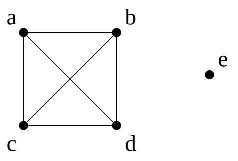
One of the more recent proposals for enhancing the evaluation of non-disjunctive programs has been the definition of backjumping and clause learning mechanisms. These techniques had been successfully employed in CSP solvers [13,14] and propositional SAT solvers [15,16] before, and were "ported" to non-disjunctive logic programming under the stable model semantics (SLP) in [1,2], resulting in the system Smodels$_{cc}$.

In this paper we address two questions:

▶ How can backjumping be generalized to disjunctive programs?

▶ Is backjumping without clause learning effective?

Why Backjumping? As for an intuition about the value of backjumping, consider the following instance of the 3COLORABILITY problem:



Note that vertex e is not on any edge.

$$edge(a,b).\ edge(a,c).\ edge(a,d).$$
$$edge(b,d).\ edge(c,b).\ edge(c,d).$$
$$vertex(a).\ vertex(b).\ vertex(c).$$
$$vertex(d).\ vertex(e).$$

Informally, an evaluation could now proceed as follows,[2] as sketched in Fig. 1. First, assume $col(a,red)$ to hold. As a consequence $col(b,red)$, $col(c,red)$, $col(d,red)$ must be false (otherwise the integrity constraint would be violated), while $col(a,blue)$ and $col(a,green)$ must be false, because they occur in the only rule which can support $col(a,red)$. Next, assume that $col(e,green)$ holds. No consequences are entailed in this case. Now, assume $col(b,green)$ to hold. As a consequence $col(c,green)$, $col(d,green)$ must be false, in order to satisfy the integrity constraint, and $col(b,blue)$ must be false because it occurs in the only rule which can support $col(b,green)$. In turn, $col(c,blue)$ and $col(d,blue)$ have to hold, as both occur in rule heads in which all other atoms are false (and the rule body is true). This, however, causes a contradiction, because both nodes on the edge (c,d) have the same color, and the respective integrity constraint is not satisfied. The reason for this contradiction is the first and third choice (*col(a,red)* and *col(b,green)*), while the second choice (*col(e,green)*) is not connected to it.

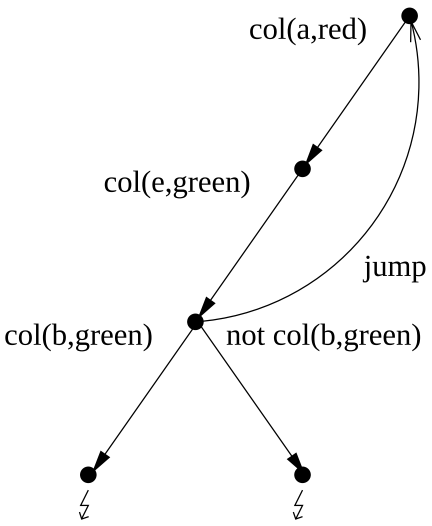


Fig. 1. Computation tree for 3COLORABILITY example.

Now, trying the complement of the last choice, $col(b,green)$, to be false, entails $col(b,blue)$ to hold, and as a consequence $col(c,blue)$, $col(d,blue)$, must be false (because of constraints), while $col(c,green)$, $col(d,green)$ must hold. Again, both nodes on the edge (c,d) would get the same color, and the respective integrity constraint is not satisfied. The reason for the contradiction is the first and third choice, as before.

We have now identified an inconsistent subtree, the reasons for the inconsistency being the first choice. Now, when going back, it does not make sense to try the complement of the second choice, as eventually the same contradiction will arise: $col(b,green)$ must be-

[2]See Section 4 for a precise description of the computation.

come true or false at some point, triggering the same contradictions. Therefore we can backjump to the closest reason of the inconsistent subtree.

In sum, once the above inconsistency arises, an algorithm based on (chronological) backtracking, tries the complement of the second choice ($col(e, green)$), making further choices and a lot of useless computations all leading to the inconsistencies encountered before. Backjumping, instead, allows us to jump directly to the source of the inconsistency ($col(a, red)$), reducing the search space significantly.

Contributions. Backjumping notions have first been studied for constraint solving, have been applied successfully for SAT solving and have recently been ported to SLP in [1,2]. In this paper we first present a generalization of these approaches 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 can then be exploited for effective backjumping. Special attention is paid to peculiarities of the disjunctive setting. We also describe the implementation of these techniques in the DLV system, the state-of-the-art SDLP 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 an experimentation activity. We have tested the impact of backjumping both with and without the employment of the lookahead (see Section 5), on random 3SAT instances, Σ_2^P-hard QBF, and some structured SAT instances.

The full picture resulting from the experiments is very positive:

- On SigmaP2-hard QBF, the Backjumping technique (BJ) reduces the number of choice points significantly. Such a reduction implies also relevant time-gains in the program evaluation. Both the reduction of choice points and the time-gain are observed even if lookahead is employed.
- The backjumping technique has a positive impact also on the evaluation of structured 3SAT instances. Choice points are reduces sensibly, and a time-gain is obtained, both with and without the lookahead.
- On random 3SAT instances, the backjumping technique brings a reduction of the search space (fewer choice points), if lookahead is not employed. This reduction is compensated by the overhead brought by the reason calculus, but such an overhead does not overcome the gain: the two versions (with/without backjumping) essentially show the same performance. If lookahead is employed, there is no cut of the search space in this case; but the overhead in computation-time, which is brought by the reason calculus, is negligible.

In sum, the results of the experiments let us conclude the following:

- Backjumping is preferable to the version without backjumping, independently of the heuristic employed in DLV.
- Backjumping without clause learning can be effective.
- Even in cases, in which the search space is not pruned by backjumping, the overhead is negligible.

The organization of the paper is as follows. In Section 2 we review the syntax and semantics of SDLP, and recall some of its properties. In Section 3, the computational core of DLV is presented, which is extended in Section 4 by a suitable backjumping method. In Section 5 we report on the benchmarks performed to asses the impact of this backjumping method. Finally, in Section 6 we draw out conclusions and outline future work.

2. Disjunctive Logic Programming

In this section, we provide a brief introduction to the syntax and semantics of Disjunctive Logic Programming; for further background see [17,18].

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[3] and $n \geq 0$, $m \geq k \geq 0$. A literal is either an atom a or its default negation $\mathrm{not}\ a$. Given a rule r, let $H(r) = \{a_1, ..., a_n\}$ denote the set of head literals, $B^+(r) = \{b_1, ..., b_k\}$ and $B^-(r) = \{\mathrm{not}\ b_{k+1}, ..., \mathrm{not}\ b_m\}$ the set of pos-

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

itive and negative body literals, resp., and $B(r) = B^+(r) \cup B^-(r)$. the set of body literals.

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, $\text{not}.L = \{\text{not}.l \mid l \in L\}$.

For example consider the following program:

$r_1: \ a(X) \ \texttt{v} \ b(X) \ \texttt{:-} \ c(X,Y), d(Y), \text{not}\ e(X).$
$r_2: \ \ \texttt{:-} \ c(X,Y), k(Y), e(X), \text{not}\ b(X)$
$r_3: \ m \ \texttt{:-} \ n, o, a(1).$
$r_4: \ c(1,2).$

r_1 is a disjunctive rule s.t. $H(r_1) = \{a(X), b(X)\}$, $B^+(r_1) = \{c(X,Y), d(Y)\}$, and $B^-(r_1) = \{e(X)\}$; r_2 is a constraint s.t. $B^+(r_2) = \{c(X,Y), k(Y), e(X)\}$, and $B^-(r_2) = \{b(X)\}$; r_3 is a ground positive (non-disjunctive) rule s.t. $H(r_3) = \{m\}$ $B^+(r_3) = \{n, o, a(1)\}$, and $B^-(r_3) = \emptyset$; r_4 is a fact (note that `:-` is omitted).

2.2. Semantics

The semantics of a disjunctive logic program is given by its stable models [19], which we briefly review in this section.

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 stable models using its ground instantiation $\mathcal{P}$ in two steps: First we define the stable models of positive programs, then we give a reduction of general programs to positive ones and use this reduction to define stable models 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 $B_{\mathcal{P}}$.[4] 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 a *stable model* for a positive program $\mathcal{P}$ if its positive part is minimal w.r.t. set inclusion among the models of $\mathcal{P}$.

For example, consider the positive programs

$$\mathcal{P}_1 = \{a \vee b \vee c. \ ; \ \ \texttt{:-}\ a.\}$$
$$\mathcal{P}_2 = \{a \vee b \vee c. \ ; \ \ \texttt{:-}\ a. \ ; \ \ b \ \texttt{:-}\ c. \ ; \ \ c \ \texttt{:-}\ b.\}$$

The stable models of $\mathcal{P}_1$ are $\{b, \text{not}\ a, \text{not}\ c\}$ and $\{c, \text{not}\ a, \text{not}\ b\}$, while $\{b, c, \text{not}\ a\}$ is the only stable model of $\mathcal{P}_2$.

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

A stable model of a general program $\mathcal{P}$ is a model X of $\mathcal{P}$ such that X is a stable model of $\mathcal{P}^X$.

Given the (general) program

$$\mathcal{P}_3 = \{$$
$$a \vee b \ \texttt{:-}\ c. \ ;$$
$$b \ \texttt{:-}\ \text{not}\ a, \text{not}\ c. \ ;$$
$$a \vee c \ \texttt{:-}\ \text{not}\ b.$$
$$\}$$

and the interpretation $I = \{b, \text{not}\ a, \text{not}\ c\}$, the reduct $\mathcal{P}_3^I$ is $\{a \vee b \ \texttt{:-}\ c., b.\}$. I is a stable model of $\mathcal{P}_3^I$, and for this reason it is also a stable model of $\mathcal{P}_3$. Now

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

consider $J = \{a, \text{not } b, \text{not } c\}$. The reduct $\mathcal{P}_3^J$ is $\{a \vee b \text{ :- } c. \ ; \ a \vee c.\}$ and it can be easily verified that J is a stable model of $\mathcal{P}_3^J$, so it is also a stable model of $\mathcal{P}_3$.

2.3. *Some SDLP 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 $r \in ground(\mathcal{P})$, i.e. 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 a stable model of a program $\mathcal{P}$, then all atoms in M are supported [20,21,22].

An important property of stable models is related to the notion of *unfounded set* [23,21]. 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[5]. Given $I \in \mathbf{I}_{\mathcal{P}}$, let $\mathcal{GUS}_{\mathcal{P}}(I)$ (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 a stable model of $\mathcal{P}$ iff $\text{not}.M = \mathcal{GUS}_{\mathcal{P}}(I)$ [21].

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 positive 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.[6]

As an example, consider the programs

$$\mathcal{P}_4 = \{a \vee b. \ ; \ c \text{ :- } a. \ ; \ c \text{ :- } b.\}$$
$$\mathcal{P}_5 = \mathcal{P}_4 \cup \{d \vee e \text{ :- } a. \ ; \ d \text{ :- } e. \ ; \ e \text{ :- } d, \text{not } b.\}.$$

The dependency graph $DG_{\mathcal{P}_4}$ of $\mathcal{P}_4$ is depicted in Figure 2 (a), while the dependency graph $DG_{\mathcal{P}_5}$ of $\mathcal{P}_5$ is depicted in Figure 2 (b).

[5]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 [21]).

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

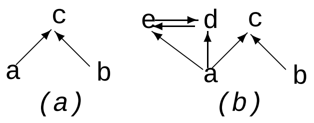


Fig. 2. *Graphs (a) $DG_{\mathcal{P}_4}$, and (b) $DG_{\mathcal{P}_5}$*

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 $DG_{\mathcal{P}}$ [24].

Considering the previous example, the dependency graphs given in Figure 2 reveal that program $\mathcal{P}_4$ is HCF and that program $\mathcal{P}_5$ is not HCF, as rule $d \vee e \text{ :- } a.$ 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.

3. Model Generation in DLV

In this section, we briefly describe the computational process performed by the DLV system [21,25] to compute stable models, which will be used for the experiments. Note that, other SDLP and SLP systems like Smodels [26,27] employ a very similar procedure.

In general, a logic program $\mathcal{P}$ contains variables. The computational step of an SDLP 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 stable models as $\mathcal{P}$ [28].

3.1. *Main Model Generation Procedure*

The nondeterministic part of the computation is 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 choice points 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 [25]. Note also that the version described here computes one stable model for simplicity, however modifying it to compute all or n stable models 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 IsUnfoundedFree(I);
    Select an undefined atom $A$ using a heuristic;
    if ( MG ( $I \cup \{A\}$ ) then return true;
        else return MG ( $I \cup \{\text{not } A\}$ ); };
```

Fig. 3. Computation of Stable Models

Roughly, the Model Generator produces some "candidate" stable models. Each candidate I is then verified by the function IsUnfoundedFree(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. This part of the computation is also referred to as *model* or *stability checker*.

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 a stable model, then the function returns true and sets I to the computed stable model; 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 SDLP for making further inferences (e.g., it uses the knowledge that every stable model is a minimal model).

DetCons(I) computes the deterministic consequences of I, and will be described in more detail in Section 3.2. 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 employs by default a so called *lookahead* heuristic [29,30] and an efficient DetCons implementation [31,32].

In a lookahead heuristic, each possible choice literal is tentatively assumed, its consequences are computed, and some characteristic values on the result are recorded. Based on these values, the choice is determined. We will not describe the method here further, as it is not connected to backjumping, and refer to [29,30] for details.

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

In Section 4, 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.

3.2. *DetCons*

As previously pointed out, the role of DetCons is similar to the Boolean Constraint Propagation (BCP, often referred to as *unit propagation*) 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 inference rules. Those rules combine an extension of the Well-founded operator for disjunctive programs with a number of techniques based on SDLP program properties. We will not define these rules or their implementation in detail here, as they are not a novelty of this paper, and refer to [31,32] for their precise definitions and implementation.

While the full implementation of DetCons involves four truth values (apart from true, false, and undefined, there is also "must be true"), we treat "must be true" as true in this description for simplicity, 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 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 [32].

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.

4.1. Backjumping by Example

Consider the following program

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

and suppose that the search tree is as depicted in Fig. 4.

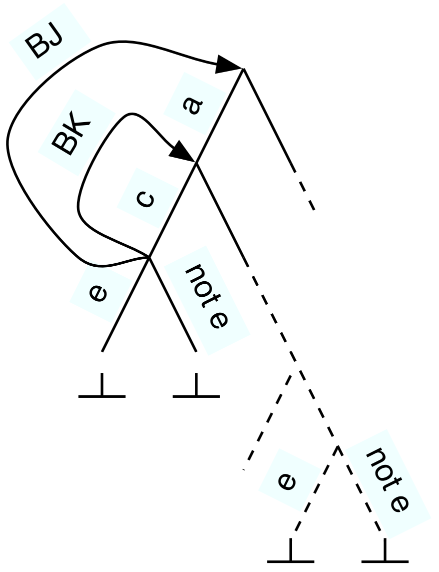


Fig. 4. Backtracking vs Backjumping.

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 choice point, in this case inverting the truth value of c (cf. the arc labelled BK in Fig. 4).

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. 4, this means that in the whole subtree below the arc labelled a 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 a to be true (see the arc labelled BJ in Fig. 4). In other words, if we know the "reasons" of an inconsistency, we can backjump directly to the closest choice that caused the inconsistent subtree.

4.2. Reasons for Literals

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

We start by reviewing the intuition of reason of a literal (representing a truth value of the literal's atom). A rule $a \text{ :- } b, c, \text{not } d.$ can give rise to the following propagation: If b and c are true and d is false in the current partial interpretation, then a is derived to be true (by Forward Propagation). In this case, we say that a is true "because" b and c are true and d is false.

More generally, the reason of a derived literal consists of the reasons of those literals that entail its truth. While for Forward Propagation it is rather clear which literals entail the derived one, this is somewhat more intricate for other propagations. However, there is one way for a literal to become true unconditionally, i.e. no other literals entail its truth: These are the *chosen* literals which become true by virtue of one of the recursive invocation in the last two lines of MG in Figure 3. In this case, their only reason is their choice.

The only elementary reasons are therefore the *chosen* literals; all other reasons are aggregations of reasons of other literals. There are also cases in which literals are unconditionally true, for example atoms occurring in facts (rules with singleton head and empty body). Since at any point during the computation there is a unique chosen literal at any recursion level, we may identify the reason of a chosen literal by an integer number (starting from 0) representing its recursion level. Reasons of derived literals are then (possibly empty) collections of integers.

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 represent the set of choices entailing l. Therefore, for any chosen literal c, $|R(c)| = 1$ holds, while for any derived (i.e., non-chosen) literal n, $|R(n)| \geq 1$ holds. For instance, if $R(l) = \{1, 3, 4\}$, then the literals chosen at recursion levels 1,3 and 4 entail l.

4.3. *Determining Reasons for Derived Literals*

In order to define more formally what reasons for derived (non-chosen) literals are — this task is often referred to as *reason calculus* — we need some preliminary notions.

Given a rule r, a "satisfying literal" is either a true head atom or a false body literal in r. Rule r is satisfied iff it has a satisfying literal. Note that a satisfied rule can have more than one satisfying literals. Also note that any satisfying literal is either a chosen or a derived literal, so we may assume that its reason is known.

Let us next define an ordering $\prec_r$ among satisfying literals of a given rule r, which is basically a lexicographic order over the numerically ordered integers of the respective reasons. We first give two technical definitions: $R_k(l)$ denotes the reason of l without the k greatest integers, while $MAX_k(l)$ gives the k-th integer of $R(l)$ in descending order (or -1 if $|R(l)| < k$).

$$R_k(l) = \begin{cases} R(l), & k = 0 \\ R_{k-1}(l)\backslash\{max(R_{k-1}(l))\}, & k > 0 \end{cases}$$

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

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) $MAX_1(s_1) < MAX_1(s_2) \wedge MAX_1(s_1 \neq -1 \wedge MAX_1(s_2) \neq -1$
(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
$MAX_k(s_1) < MAX_k(s_2) \wedge MAX_k(s_1) \neq -1 \wedge MAX_k(s_2) \neq -1$
(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 $s_1, \ldots, s_n$ be 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 a cancelling assignment of a rule r represents the reason corresponding to the earliest choice 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. We would like to point out that we did not introduce any novel inference rules in this work, but rather we extend existing ones by the reason calculus.

Forward Inference
We have already discussed that case informally; if all body literals are true, all head atoms but one (which is undefined) are false, then we infer the truth of the only undefined head atom. The reason for this head atom is the set consisting of the union of the reasons for all the other literals in the rule.

More formally, 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 $a_i \in I$. We define $R(a_i)$ as $\bigcup_{a \in H(r)\setminus\{a_i\}} R(\text{not } a) \cup \bigcup_{l \in B(r)} R(l)$.

For example, consider the following program:

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

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
In this case we derive negative information: If for some undefined atom all of the rules, in which it occurs in the head, are satisfied w.r.t. I, then derive this atom to be false. So the reason for the derived literal is that all of the rules, in which it occurs in the head, are satisfied w.r.t. I. As motivated above, the reason for a rule being satisfied is its cancelling assignment. The reason of the literal is hence the union of the respective cancelling assignments.

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 $\text{not}.a \in I$. We define $R(a)$ as $\bigcup_{r : a \in H(r)} R_r$, where R_r is a cancelling assignment of rule r.

For example, consider the following subprogram:

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

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
Here we exploit the fact that each true atom in a stable model must have at least one supporting rule. If an atom is true and has only one supporting rule left, we infer the truth of all body literals and the falsity of all other head atoms of that rule. Note that rules which do not support this atom are satisfied, so they have a cancelling assignment. The reason for all literals derived in this way consist of the fact that a is true and that all other possibly supporting rules are satisfied, hence the reason for the atom unified with all cancelling assignments of the other rule.

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) $\exists c' \in H(r')$: $c' \neq a \wedge c' \in I$, then for each $c \in H(r)$ s.t. $c \neq a$ and $\text{not } c \notin I$ infer $\text{not } c \in I$, and for each $b \in B(r) \setminus I$ infer $b \in I$. For each derived literal l of r, $R(l) = R(a) \cup \bigcup_{r' : a \in H(r') \wedge r' \neq r} R_{r'}$.

For example, considering the following program

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

suppose that $I = \{a, f, g, \text{not } k\}$ 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
In order to enforce satisfaction of a rule, if the head of a rule is false, and all but one (undefined) body literals are true, we infer that the remaining undefined literal must be false. In this case, the reason for the falsity of that atom is – similar to the case of Forward Propagation – the union of the reasons for head atoms to be false and the other body literals to be true.

Given a rule r such that (i) $\forall a \in H(r) : \text{not}.a \in I$, (ii) $\exists l \in B(r) : l \notin I \wedge \text{not}.l \notin I$, (iii) $\forall b \in B(r) \setminus \{l\} : b \in I$, then infer $\text{not}.l \in I$. We set $R(l) = \bigcup_{a \in H(r)} R(\text{not } a) \cup \bigcup_{b \in B(r) \setminus \{l\}} R(b)$.

Consider the following program,

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

and suppose that $I = \{\text{not } a, \text{not } b, d, e\}$, 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
In this case we use the result that any stable model does not contain any atom which is in some unfounded set (w.r.t. the stable model). If we determine that some set of atoms is unfounded w.r.t. the current interpretation, all of the atoms in this set can be derived as false. The reason for some atom to be in an unfounded set is that all of the rules, in the head of which it occurs, are either satisfied or contain some atom of the unfounded set in its positive body. In the former case the reason is obviously the cancelling assignment, while in the latter case, there is no reason other than the presence of the unfounded set itself. So whenever there is a reason for the satisfied rules to be satisfied, this unfounded set will exist. Therefore unsatisfied rules with unfounded positive body do not contribute to the reason.

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 other element from X).

For example, consider the following program:

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

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 $R^*_{r_3} = R^*_{r_4} = \emptyset$.

4.4. Reasons for Inconsistencies

So far, we have described what reasons for literals are and how to determine them. We will now turn to how to exploit them during the computation. As motivated before, we will use reason information when inconsistencies occur, in order to understand what assumptions have to be changed in order to avoid the inconsistency, and what other assumptions do not have any influence on the inconsistency.

In DLV, we can isolate two main sources of inconsistency:

1. Deriving conflicting literals, and
2. failing stability checks.

Of these two, the second one is particular for SDLP, while the first one is the only source for inconsistencies in SAT and non-disjunctive SLP.

Deriving conflicting literals means, in our setting, that DetCons determines that an atom a and its negation $\mathrm{not}\ a$ should both hold. In this case, the reason of the inconsistency is – rather straightforward – the combination of the reasons for a and $\mathrm{not}\ a$: $R(a) \cup R(\mathrm{not}.a)$. Obviously, this inconsistency reason does not depend on the inference rules used when determining the inconsistency.

Reconsidering the example in Section 4.1, the reason of the first inconsistency is the union of the reasons for g, $R(g) = \{0, 2\}$, and $\mathrm{not}\ g$, $R(\mathrm{not}\ g) = \{0, 2\}$, which is the set $\{0, 2\}$. So only the literals chosen at levels 0 and 2 give reason to the inconsistency, while the literal chosen at level 1 (c) is detached from the inconsistency.

As mentioned above inconsistencies from failing stability checks are a peculiarity of SDLP. This situation occurs if the function IsUnfoundedFree(I) of Figure 3 returns false. Intuitively, this means that the current interpretation (which is guaranteed to be a model) is not stable. From [21] we know that this interpretation is not unfounded-free, i.e. some positively interpreted atom is in an unfounded set w.r.t. this interpretation.

This situation is similar to the well-founded negation operator described above. The difference is that in case of a failed stability check, some unfounded atoms are already true in the interpretation, while they are normally undefined in the case of well-founded negation. Note that with the default computation strategy employed in DLV, failed stability checks will be due to some non-HCF subprogram, as otherwise the well-founded negation operator would have triggered before.

The reason for such an inconsistency is therefore based on an unfounded set, which has been determined during IsUnfoundedFree(I). Given such an unfounded set, the reason for the inconsistency is composed of the cancelling assignments for satisfied rules which contain unfounded atoms in their head. As with well-founded negation, unsatisfied rules with unfounded atoms in their head do not contribute to the reason.

Let S be a non-HCF subprogram of P, I be an interpretation, and X be an unfounded set of S w.r.t. I, such that $I \cap X \neq \emptyset$. The inconsistency reason is determined as follows: $\bigcup_{r \in S : a \in X \wedge a \in H(r)} R^*_r$, 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 other element from X).

4.5. Using Inconsistency Reasons for Backjumping

When inside MG (cf. Figure 3) some inconsistency is detected (in DetCons or IsUnfoundedFree), we analyze the inconsistency reason, and can go directly to the greatest level in the inconsistency reason. Going to any level in between (if it exists) would indeed trigger the encountered inconsistency again and again. It is worth noticing that when an inconsistency is encountered during DetCons, the inconsistency reason will always contain the last but one level, amounting to simple backtracking.

The inconsistency reasons can be further exploited: Whenever a recursive invocation of MG returns false, we know that there has been an inconsistency in this branch, and we can re-use the inconsistency reasons determined in it for the inconsistency reason of the respective branch, by stripping off all recursion levels which are greater than the current one. This is semantically correct, as in the presence of the remaining reasons, an inconsistency will definitely occur. If at any level, both recursive invocation return false, we know that the entire subtree is inconsistent. The reason for this tree to be inconsistent are then the union of the two inconsistency reasons of the branches, minus the current level (as the inconsistency does not depend on the choice of the current level). We can then continue by going directly to the greatest level in this inconsistency reason.

The case where these techniques allow for going directly to a level, which is not the previous recursion level, is frequently referred to as *backjumping*, in contrast to *backtracking*.

4.6. Model Generator with Backjumping

In this section we describe MGBJ (shown in Fig. 5), a modification of the MG function (as described in section 3), which is able to perform non-chronological backtracking, as described in Section 4.5.

It extends MG by introducing additional parameters and data structures, in order to keep track of reasons and to control backtracking and backjumping. In particular, two new parameters IR and bj_level are introduced, which hold the inconsistency reason of the subtree of which the current recursion 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 MGBJ and used for holding the current recursion level, and the reasons for the positive and negative recursive branch, respectively.

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

    bj_level ++;
    int curr_level = bj_level;

    if ( ! DetConsBJ ( I, IR )
      return false;
    if ( “no atom is undefined in I” )
      if IsUnfoundedFreeBJ( 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 ( MGBJ( $I \cup \{A\}$, posIR, bj_level )
      return true;
    if (bj_level $<$ curr_level)
      IR = posIR;
      return false;

    bj_level = curr_level;
    $R(\text{not } A)$ = { curr_level };
    if ( MGBJ ( $I \cup \{\text{not } A\}$, 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. 5. Computation of stable models with backjumping

Initially, the MGBJ function is invoked with I set to the empty interpretation, IR set to the empty reason, and bj_level set to -1 (but it will become 0 immediately). Like the MG function, if the program $\mathcal{P}$ has a stable model, then the function returns true and sets I to the computed stable model; otherwise it returns false. Again, it is straightforward to modify this procedure in order to obtain all or up to n stable models. Since these modification gives no additional insight, but rather obfuscates the main technique, we refrain from presenting it here.

MGBJ first calls DetConsBJ, an enhanced version of the DetCons procedure. In addition to DetCons, DetConsBJ computes the reasons of the inferred literals, as described in Section 4.3. Moreover, if at some point an inconsistency is detected (i.e. the complement of a true literal is inferred to be true), DetConsBJ builds the reason of this inconsistency and stores it in its new, second parameter IR before returning false. If an inconsistency is encountered, MGBJ immediately returns false and no backjumping is done. This is an optimization, because it is known that the inconsistency reason will contain the previous recursion level. There is therefore no need to analyze the levels.

If no undefined atom is left, MGBJ invokes IsUnfoundedFreeBJ, an enhanced version of IsUnfoundedFree. In addition to IsUnfoundedFree, IsUnfoundedFreeBJ computes the inconsistency reason in case of a stability checking failure, and sets the second parameter IR accordingly. If this happens, it might be possible to backjump, and 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 MGBJ. If the stability check succeeded, we just return true.

Otherwise, an atom A is selected according to a heuristic criterion. We set the reason of A to be the current recursion level and invoke MG recursively, using $posIR$ and bj_level to be filled in case of an inconsistency. If the recursive call returned true, MGBJ just returns true as well. If it returned false, the corresponding branch is inconsistent, $posIR$ holds the inconsistency reason and bj_level the recursion level to backtrack or backjump to.

Now, if bj_level is less than the current level, this indicates a backjump, and we return from the procedure, setting the inconsistency reason appropriately before. If not, then we have reached the level to go to. We set the reason for $\text{not } A$, and enter the second recursive invocation, this time using $negIR$ and reusing bj_level (which is reinitialized before).

As before, if the recursive call returns true, MGBJ immediately returns true also, while if it returned false, we check whether we backjump, setting IR and immediately returning false. If no backjump is done, this instance of MGBJ is the root of an inconsistent subtree, and we set its inconsistency reason IR to the union of $posIR$ and $negIR$, deleting all integers which are greater or equal than the current recursion level (this is done by the function trim), as described in Section 4.5. We finally set bj_level to the maximum of the obtained inconsistency reason (or 0 if the set is empty) and return false.

The actual implementation in DLV is slightly more involved, but only due to technical details. First of all, as mentioned above, the procedure was extended in order to be able to compute all stable models. We also had to deal with the additional truth value "must be true" (which is handled like true for our purposes). Furthermore, in DLV the computation tree is not really binary, but is rather a collapsed binary tree. There is, however a 1-to-1 correspondence between this collapsed binary tree and the binary tree presented here. Indeed in our implementation, we construct a virtual binary tree in order to keep track of the correct levels. Since we do not believe that these technical issues give any particular insight, but are instead rather lengthy in description, we have opted to not include them.

5. Benchmarks

In order to evaluate the backjumping technique described in Section 4, we have implemented it as an experimental extension of the DLV system. Concerning experiments, judging from results on SAT (e.g. in [33]), backjumping has the greatest impact on large, structured problem instances, so we have studied such instances, which have been described in [34]. Since we want our tool to be efficient for arbitrary input, we have also considered randomly generated hard 3SAT problem instances. These can be seen as important corner cases that the method should be able to deal with in an efficient way. We have also experimented with randomly generated 2QBF instances, which are characteristic for SDLP (they cannot be represented in SAT or non-disjunctive SLP under standard complexity assumptions).

Next, we first describe the compared systems, the benchmark problems and instances and finally we report and discuss the results of the experiments.

5.1. Compared Systems

We will now describe the systems that we have used in the experimentation. Our principal comparison is of course between the DLV system without and with the backjumping technique described in Section 4.

But there is another parameter, which is important in this respect. The choice of the heuristic function has a strong impact on the effectiveness of the backjumping technique (noted also for SAT, cf. [33]), and therefore we consider both systems first with a weak and then with a strong heuristic. In particular, the weak heuristic basically amounts to a random choice strategy. The strong heuristic employs a lookahead technique, that is, DetCons is invoked on each possible choice atom, and some values of the result are collected. These values are then used to choose the "best" atom. For details of this heuristic function, we refer to [29,30]. The important aspect is that inconsistencies can be encountered during the lookahead. This is like having made one choice, which immediately leads to an inconsistency. Our implementation treats this scenario as if a choice has actually been made.

In the sequel, we will refer to systems employing the weak heuristic as *without lookahead*, and to systems with the strong heuristic as *with lookahead*. It should be noted that choices made by the strong heuristic are less likely to lead into inconsistent branches, and so the gain by using backjumping is more limited than with a weaker heuristic. This has already been discussed at length in the SAT community, we again refer to [33] for an overview.

We will thus deal with the following four versions of the SDLP system DLV.

$STDN$ The original DLV system without lookahead. It uses the standard implementation of MG, DetCons, and IsUnfoundedFree, without reason computation, and employs the weak heuristic.

BJN This system is DLV enhanced by the backjumping technique using MGBJ, DetConsBJ, and IsUnfoundedFreeBJ, as described in Section 4.6, without lookahead.

$STDL$ The original DLV system with lookahead. It uses the standard implementation of MG, DetCons, and IsUnfoundedFree, without reason computation, and employs the strong heuristic. This is default setting for official DLV releases.

BJL The final system is DLV enhanced by the backjumping technique using MGBJ, DetConsBJ, and IsUnfoundedFreeBJ, as described in Section 4.6, this time with lookahead.

Our experiments have been performed on a 1.400 MHz Pentium 4 machine machine with 256K of Level 2 Cache and 256MB of RAM, running SuSE Linux 9.0. The binaries were generated with GCC 3.3.1

(shipped with the system). We have allowed at most one hour of execution time for each instance. For those tests, where there are multiple instances per instance size, the experimentation was stopped (for each system) at the size at which some instance exceeded this time limit.

5.2. Benchmark Problems

Boolean Satisfiability. 3SAT is one of the best researched problems in AI and generally used for solving many other problems by translating them to 3SAT, solving the 3SAT problem, and transforming the solution back to the original domain:

Let Φ be a propositional formula in conjunctive normal form (CNF) $\Phi = \bigwedge_{i=1}^{n}(d_{i,1} \vee \ldots \vee d_{i,3})$ where the $d_{i,j}$ are classical literals over the propositional variables $x_1, \ldots, x_m$. Φ is satisfiable, iff there exists a consistent conjunction I of literals such that $I \models \Phi$.

3SAT is a classical NP-complete problem and can be easily represented in SDLP as follows: For each propositional variable x_i ($1 \leq i \leq$m), we add the following rule which ensures that we either assume that variable x_i or its complement nx_i true: $x_i \,\mathtt{v}\, nx_i.$ For each clause $d_1 \vee \ldots \vee d_3$ in Φ we add the constraint `:-` $\text{not } \bar{d}_1, \ldots, \text{not } \bar{d}_3.$ where $\bar{d}_i$ ($1 \leq i \leq 3$) is x_j if d_i is a positive literal x_j, and nx_j if d_i is a negative literal $\neg x_j$.

Our test in this domain include some randomly generated 3SAT problems and "structured" instances (circuit verification benchmarks) from the the Superscalar Suite SSS.1.0 of Miroslav Velev, cf. [34].

We have randomly generated 20 3SAT instances for each problem size by using a tool by Selman and Kautz, which is available at `ftp://ftp.research.att.com/dist/ai/` . The number of clauses for each generated instance is 4.3 times the number of propositional variables (in order to generate hard instances). The SSS.1.0 instances, available in DIMACS format were easily converted in an equivalent SDLP program as indicated above.

All input files used for the benchmarks on random instances are available on the web at `http://www.mat.unical.it/leone/backjumping/aicom.tar.gz` , while the SSS.1.0 instances can be found at `http://www.ece.cmu.edu/~mvelev` .

Quantified Boolean Formulas. To asses the impact of backjumping on Σ_2^P-complete problems we used "$\exists\forall$" Quantified Boolean Formulas (2QBF)[7], which have already been used in the past for benchmarking SDLP systems [9,12].

The problem here is to decide whether a quantified Boolean formula (QBF) $\Phi = \exists X \forall Y \phi$, where X and Y are disjoint sets of propositional variables and $\phi = C_1 \vee \ldots \vee C_k$ is a 3DNF formula over $X \cup Y$, is valid. The transformation from 2QBF to disjunctive logic programming we use here has been given in [9], based on a reduction presented in [36]. The propositional disjunctive logic program $\mathcal{P}_\phi$ produced by the transformation contains the following rules:

$$
\begin{array}{rl}
\multicolumn{2}{l}{t(true). \quad f(false).} \\
t(X) \,\mathtt{v}\, f(X) & \texttt{:-}\ exists(X). \\
t(Y) \,\mathtt{v}\, f(Y) & \texttt{:-}\ forall(Y). \\
w & \texttt{:-}\ term(X, Y, Z, Na, Nb, Nc), \\
 & \quad t(X), t(Y), t(Z), f(Na), \\
 & \quad f(Nb), f(Nc). \\
t(Y) & \texttt{:-}\ w, forall(Y). \\
f(Y) & \texttt{:-}\ w, forall(Y). \\
 & \texttt{:-}\ \text{not } w.
\end{array}
$$

Moreover, $\mathcal{P}_\phi$ contains the following facts:

- $exists(v)$, for each existential variable $v \in X$;
- $forall(v)$, for each universal variable $v \in Y$; and
- $term(p_1, p_2, p_3, q_1, q_2, q_3)$, for each disjunct $l_1 \wedge l_2 \wedge l_3$ in ϕ, where (i) if l_i is a positive atom v_i, then $p_i = v_i$, otherwise p_i= "$true$", and (ii) if l_i is a negated atom $\neg v_i$, then $q_i = v_i$, otherwise q_i="$false$".
  For example, $term(x_1, true, y_4, false, y_2, false)$, encodes the term $x_1 \wedge \neg y_2 \wedge y_4$.

The 2QBF formula Φ is valid iff $\mathcal{P}_\Phi$ has an answer set [36].

We used the benchmark instances from [9]. There, 50 hard instances per problem size were randomly generated. Accordingly with [37,38], each formula contains the same number of universal and existential variables ($|X| = |Y|$), and the number of clauses is equal to the number of variables ($|X| + |Y|$). The input files used for the benchmarks are available on the web at `http://www.dlvsystem.com/examples/tocl-dlv.zip`.

[7]2QBF is well-known to be a Σ_2^P-complete problem see [35].

5.3. *Experimental Results*

In this section we report the obtained results. We will first report on the case without lookahead (i.e. using the weak heuristic), followed by the results with lookahead (i.e. using the strong heuristic).

5.3.1. *Results without Lookahead*

We start reporting on the random 3SAT instances. A critical internal measure is the number of choice points, these choice points correspond to the number of times MG or MGBJ have been invoked. If backjumping occurs, there will be fewer choice points. One could also count the number of backjumps, but this measure would be bogus, as obviously not only the number, but also the length of the backjumps are important. The number of choice points, however, is a direct measure of the structural savings brought about by backjumps. Fig. 6 shows the average (left) and maximum (right) number of choice points per instance size. Note that we have employed a logarithmic scale in all of our diagrams, as they measure concepts which grow exponentially. BJN scales slightly better, in the final instance size (155), BJN takes about 800000 fewer choice points than STDN on average, while the maximum number of choice points consumed for this instance size is almost 4 million more for STDN.

But having backjumping also incurs some overhead (most importantly, maintaining reasons), which can obviously not be measured in terms of choice points. So in Fig. 7, we report on the average (left) and maximum (right) execution time. We observe that the potential benefits of saved choice points is outweighed by the overhead incurred by the reason computations, but importantly, there is no slowdown.

Let us now turn to the structured satisfiability instances. Here, we have allowed two hours of execution time, and report only on those instances, which have been solved by at least one of the tested systems in the allotted time. The execution times are reported in Table 1, and choice points are reported in Table 2. We can see that STDN was not able to solve any of these instances within 2 hours. BJN, however, could solve one instance (dlx1_c), the number of explored choice points is quite impressive (about 80 millions).

Let us next turn to the final benchmark problem, 2QBF. In Fig. 8, the average (left) and maximum (right) number of choice points per instance size is reported. Compared to 3SAT, we can observe a more drastic saving with BJN with respect to STDN in choice points for 2QBF. There are two peculiar spikes in the graphs: Comparing the average and maximum graph, it becomes clear that these are due to two exceptionally hard instances occurring at sizes 44 and 56, respectively, in which fewer backjumps than in most other instances are possible. This seems to be a peculiarity of the distribution of the underlying instance data in combination with the weak heuristic.

Instance	STDN	BJN	STDL	BJL
dlx1_c	$>$2h	5464.80s	306.33s	270.51s
dlx2_cc_bug04	$>$2h	$>$2h	3.57s	2.91s
dlx2_cc_bug06	$>$2h	$>$2h	$>$2h	5498.96s
dlx2_cc_bug07	$>$2h	$>$2h	1301.40s	814.97s
dlx2_cc_bug08	$>$2h	$>$2h	1890.81s	854.00s

Table 1
Execution Time on solved SSS.1.0 SAT instances

Instance	STDN	BJN	STDL	BJL
dlx1_c	-	79989185	221925	169774
dlx2_cc_bug04	-	-	167	154
dlx2_cc_bug06	-	-	-	688145
dlx2_cc_bug07	-	-	177499	88965
dlx2_cc_bug08	-	-	295785	91394

Table 2
Choice Points on solved SSS.1.0 SAT instances

Different to 3SAT, the reduction of the search space for BJN in 2QBF is proportionally much higher, thus by far outweighing the overhead incurred by the reason computations, as can be observed in Fig. 9, in which we report average (left) and maximum (right) execution time. We note that BJN scales much better than STDN: While BJN could solve each instance up to size 80 within 1 hour each, this is only possible up to size 52 for STDN.

5.3.2. *Results with Lookahead*

Let us now turn to the versions with lookahead and a strong heuristics. Originally, we did not expect too much of this combination, as one of the conclusions in similar studies for SAT seemed to be that the combination of strong heuristics and backjumping (including clause learning) does not have advantages in general. However, as the results in this section will show, it seems that in our setting this combination works indeed well.

We start by examining the results on the randomly generated 3SAT instances. We found that in this case the number of choice points is essentially equal, so

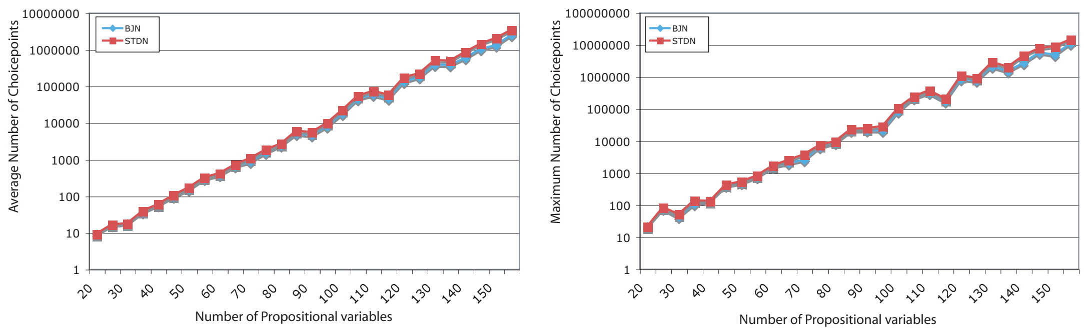


Fig. 6. Choice points on Random 3SAT instances without lookahead

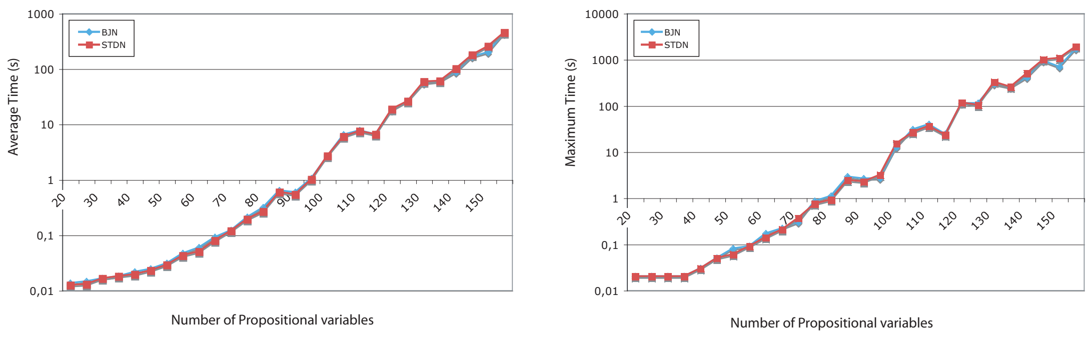


Fig. 7. Execution Times on Random 3SAT problems without lookahead

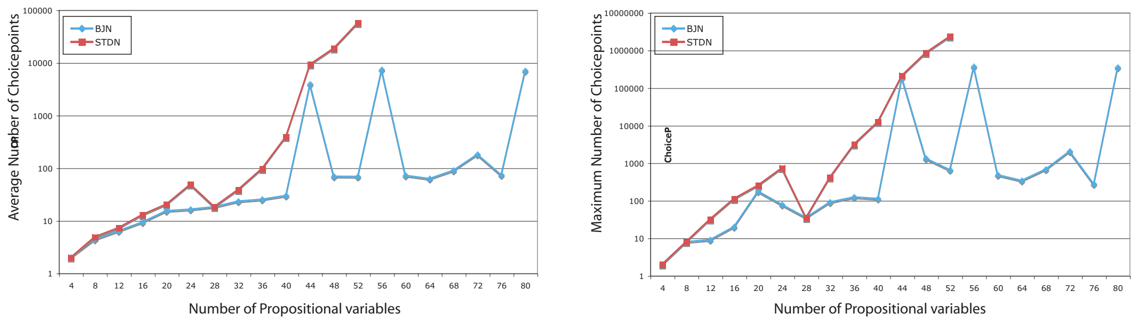


Fig. 8. Choice points on random 2QBF problems without lookahead

backjumping does basically not have any impact here (we omit this graph, as it shows just two overlapping lines). However, looking at Fig. 10, where, as usual, the average time is reported in the left graph, and the maximum time in the right one, we observe that the overhead in execution time is very small.

Looking at the results of the structured SAT instances in Tables 1 and 2, the picture is quite different: Already STDL can solve many more instances than STDN within 2 hours, but BJL manages to solve one (dlx2_cc_bug06) within 2 hours, which no other tested system could do. Also in the other examples, BJL is always the fastest system, sometimes more than twice as fast as STDL. Also the number of choice points is

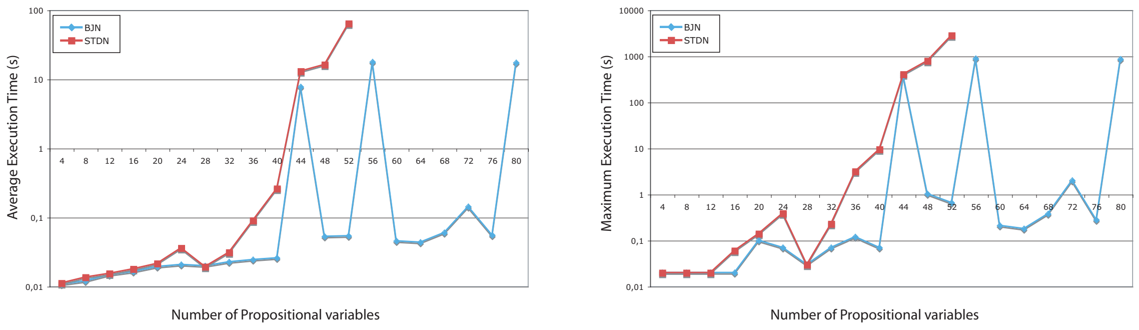


Fig. 9. Execution time on random 2QBF problems without lookahead

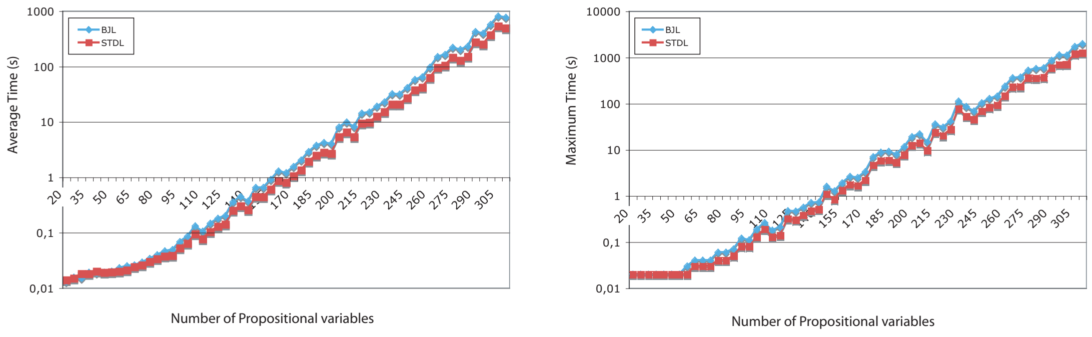


Fig. 10. Execution time on random 3SAT problems with lookahead

always drastically lower for BJL than for STDL.

Finally, we report on the experiments with 2QBF instances. Fig. 11 shows the average (left) and maximum (right) number of choice points per instance size. We see that also with lookahead, the number of choice points is cut effectively by BJL with respect to STDL. Fig. 11 reports the average (left) and maximum (right) execution time per instance size. Also here, BJL clearly has an edge over STDL. BJL also managed to solve more instances within the allotted time than STDL.

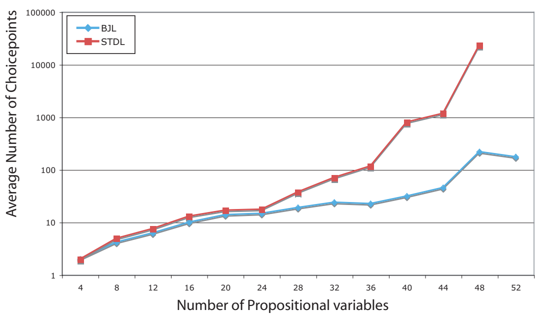
5.3.3. *Summary*

We have observed that both with and without lookahead, backjumping generally cuts the search space effectively and consumes (often dramatically) less execution time than the systems without backjumping. The only exception is 3SAT on random instances, where BJL has a mild overhead in execution time. But this minimal overhead is definitely outweighed by the big advantages BJL and BJN show on structured SAT and 2QBF instances, both with and without heuristics. A consequence of these results is that backjumping without clause learning is, at least for SDLP, effective.

6. Conclusion and Future Work

We have presented a backjumping technique for computing the stable models of disjunctive logic programs. It is based on a reason calculus and is an elaboration of the work in [1,2], but our work contains some crucial novelties and improvements: Most importantly, our framework is suitable and tailored for disjunctive programs, including novel techniques for this setting. Concerning the reason computation, our method does not incur building an implication graph, but rather store only sets of integers, which are more efficient to compute, and also give an easier handle on determining the point to jump to.

We have implemented the technique in the DLV system, and have conducted several experiments with it. In total, the backjumping technique has a very positive effect on performance in many cases, and even in cases, in which it cannot cut the search space, its overhead is negligible. Moreover, these improvements can be observed with either of two heuristic methods, which are diametrically different from each other. So we conclude that the technique for SDLP is robust with



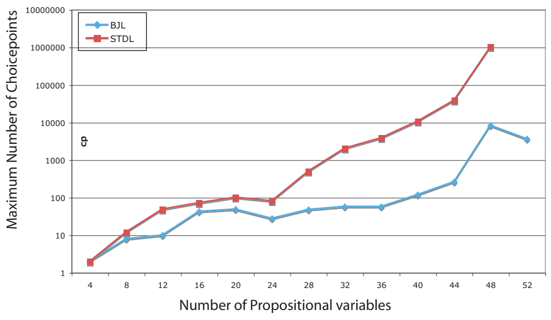
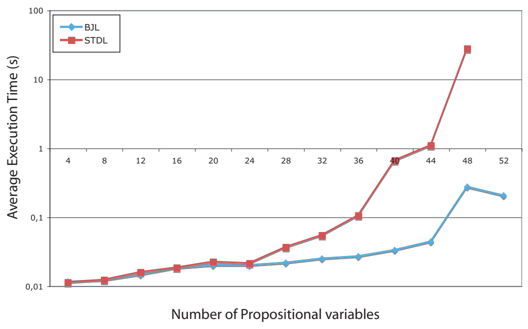


Fig. 11. Choice Points on random 2QBF problems with lookahead



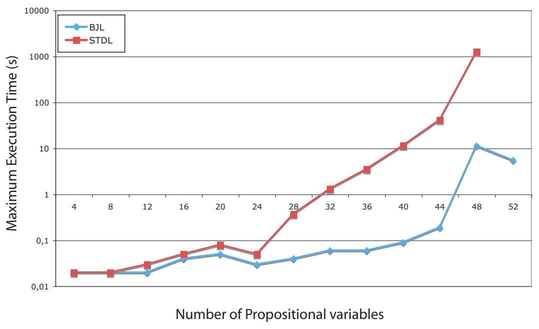


Fig. 12. Execution time on random 2QBF problems with lookahead

respect to the heuristic method, and in particular, co-operates well with a lookahead heuristic.

Our backjumping technique is very effective on structured satisfiability instances, and on randomly generated hard 2QBF instances (which cannot be solved by SAT solvers directly under standard complexity assumptions). It shows little to no impact, but also no relevant slowdown on randomly generated hard satisfiability instances.

For future work, we want to address the question whether clause learning can yield an additional gain w.r.t. backjumping in SDLP. Implementing clause learning in DLV is, however, not straightforward at all, as the DLV model generator heavily relies on the assumption that the program, on which it works, is fixed. There are several ways of overcoming this difficulty, ranging from a redesign of the data structures to the introduction of an additional structure which is dedicated to the learned clauses.

Another possibility for future work is to study the use of our reason calculus not only for stable model computation, but also for software engineering tasks. In particular, it has been observed that reasons could be profitably used for debugging SDLP programs. A program, for which some particular stable model, which has been expected by its author, does not exist, these reasons could be used to find the point in the program which forbids the existence of the expected stable model, and thus the modeling bug.

Acknowledgements

This work was supported by the European Commission under projects IST-2002-33570 INFOMIX and IST-2001-37004 WASP, and was partially supported by M.I.U.R. under project ”Sistemi basati sulla logica per la rappresentazione di conoscenza: estensioni e tecniche di ottimizzazione”. Wolfgang Faber’s work was funded by an APART grant of the Austrian Academy of Sciences.

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] Lifschitz, V.: Answer Set Planning. In Schreye, D.D., ed.: ICLP'99, Las Cruces, New Mexico, USA, The MIT Press (1999) 23–37

[4] Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM TODS **22** (1997) 364–418

[5] Rintanen, J.: Improvements to the Evaluation of Quantified Boolean Formulae. In Dean, T., ed.: IJCAI 1999, Sweden,(1999) 1192–1197

[6] Eiter, T., Gottlob, G.: The Complexity of Logic-Based Abduction. JACM **42** (1995) 3–42

[7] Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. CUP (2002)

[8] Leone, N., Rosati, R., Scarcello, F.: Enhancing Answer Set Planning. In: IJCAI-01 Workshop on Planning under Uncertainty and Incomplete Information. (2001) 33–42

[9] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL (2005) To appear.

[10] Janhunen, T., Niemelä, I., Seipel, D., Simons, P., You, J.H.: Unfolding Partiality and Disjunctions in Stable Model Semantics. Tech. Report cs.AI/0303009, arXiv.org (2003)

[11] Lierler, Y.: Disjunctive Answer Set Programming via Satisfiability. In: Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR'05, Diamante, Italy, 2005, Proceedings. LNCS 3662

[12] Koch, C., Leone, N., Pfeifer, G.: Enhancing Disjunctive Logic Programming Systems by SAT Checkers. Artificial Intelligence **15** (2003) 177–212

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

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

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

[16] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, 18-22, 2001, ACM (2001) 530–535

[17] 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

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

[19] Przymusinski, T.C.: Stable Semantics for Disjunctive Programs. NGC **9** (1991) 401–424

[20] Marek, W., Subrahmanian, V.: The Relationship between Logic Program Semantics and Non-Monotonic Reasoning. In: ICLP'89, MIT Press (1989) 600–617

[21] Leone, N., Rullo, P., Scarcello, F.: Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation. Information and Computation **135** (1997) 69–112

[22] Baral, C., Gelfond, M.: Logic Programming and Knowledge Representation. JLP **19/20** (1994) 73–148

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

[24] Ben-Eliyahu, R., Dechter, R.: Propositional Semantics for Disjunctive Logic Programs. AMAI **12** (1994) 53–87

[25] Faber, W.: Enhancing Efficiency and Expressiveness in Answer Set Programming Systems. PhD thesis, TU Wien (2002)

[26] 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

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

[28] 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

[29] Faber, W., Leone, N., Pfeifer, G.: Experimenting with Heuristics for Answer Set Programming. In: IJCAI 2001, Seattle, WA, USA,(2001) 635–640

[30] Faber, W., Leone, N., Pfeifer, G.: Optimizing the Computation of Heuristics for Answer Set Programming Systems. In: LPNMR'01. LNCS 2173

[31] Faber, W., Leone, N., Pfeifer, G.: Pushing Goal Derivation in DLP Computations. In: LPNMR'99. LNCS 1730

[32] Calimeri, F., Faber, W., Leone, N., Pfeifer, G.: Pruning Operators for Answer Set Programming Systems. In: NMR'2002. (2002) 200–209

[33] Lynce, I., Silva, J.P.M.: Building state-of-the-art sat solvers. In van Harmelen, F., ed.: Proceedings of the 15th Eureopean Conference on Artificial Intelligence (ECAI 2002), IOS Press (2002) 166–170

[34] Velev, M.N., Bryant, R.E.: Superscalar Processor Verification Using Efficient Reductions of the Logic of Equality with Uninterpreted Functions to Propositional Logic. In: Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, 27-29, 1999, Proceedings. NY, USA, (1999) 37–53

[35] Papadimitriou, C.H.: Computational Complexity. Addison-Wesley (1994)

[36] Eiter, T., Gottlob, G.: On the Computational Cost of Disjunctive Logic Programming: Propositional Case. AMAI **15** (1995) 289–323

[37] Cadoli, M., Giovanardi, A., Schaerf, M.: Experimental Analysis of the Computational Cost of Evaluating Quantified Boolean Formulae. In: AI*IA 97. Italy, (1997) 207–218

[38] Gent, I., Walsh, T.: The QSAT Phase Transition. In: AAAI. (1999)