Evaluating Backjumping for Disjunctive Logic Programming

Valutazione del Backjumping per Programmazione Logica Disgiuntiva

Wolfgang Faber Nicola Leone Marco Maratea
Francesco Ricca

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

SOMMARIO/*ABSTRACT*

In questo articolo presentiamo la tecnica del backjumping per la Programmazione Logica Disgiuntiva, prendendo spunto dai lavori svolti per solutori per Constraint e logica proposizionale. A differenza dei lavori citati, ci focalizzaremo su backjumping senza (clause) learning. Presenteremo un fondamento teorico per questa tecnica sfruttando le peculiarità della logica su cui lavoriamo. Presenteremo un calcolo delle reasons che, confrontato con metodi tradizionali, riduce le informazioni che devono essere mantenute, salvaguardando allo stesso tempo la correttezza ed efficienza della tecnica del backjumping. La suddetta tecnica è stata implementata in DLV. Per valutarne l'efficacia, abbiamo svolto diversi esperimenti sia su istanze generate casualmente che su istanze strutturate, studiando inoltre l'effetto dell'interazione tra backjumping e due diverse euristiche. I risultati degli esperimenti fanno notare come il backjumping ha effetti positivi usando entrambe le euristiche, in particolare su programmi derivanti da istanze strutturate in logica porposizionale e da altre istanze in logica booleana quantificata.

In this work we present a backjumping technique for Disjunctive Logic Programming under the Stable Model Semantics, building upon technologies developed for constraint solving and satisfiability testing. Different from most related work, we focus on backjumping without (clause) learning, providing a new theoretical framework for backjumping in our setting, exploiting its peculiarities. We present a reason calculus and associated computations, which, compared to the traditional approaches, reduce 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. We have conducted experiments on hard random and structured instances in order to assess the impact of our technique, studying the effect of the backjumping method in relation to two different heuristics. The experiments suggest that backjumping has positive effects for any of these two heuristics. In particular, we have noted a great reduction in execution time for structured propositional satisfiability and quantified boolean formula instances.

1 Introduction

SDLP. Disjunctive Logic Programming under the Stable Model Semantics (SDLP) [1], is a 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 [16].

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) [8]. 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, and can be naturally encoded in SDLP [1; 14]. By now, several systems are available, which implement SDLP: DLV [2], GnT [3], and recently cmodels-3 [4].

Main Issues. Most of the optimization work on related SDLP systems has focused on the efficient evaluation

---

[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.

[2] `http://www.dlvsystem.com/`

[3] `http://www.tcs.hut.fi/Software/gnt/`

[4] `http://www.cs.utexas.edu/users/tag/cmodels/`

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 [12]).

Among the most recent proposals for enhancing the evaluation of non-disjunctive programs, we mention the definition of backjumping and clause learning mechanisms. These techniques had been successfully employed in CSP solvers [19; 5] and propositional SAT solvers [2; 18] before, and were "ported" to non-disjunctive logic programming under the stable model semantics (SLP) in [23; 22], 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?

Contributions. In this paper we first present a generalization of the CSP and SAT approaches for backjumping 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). The reasons established by the calculus 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, on random 3SAT instances, Σ_2^P-hard QBF, and some structured SAT instances. The results of the experiments let us conclude the following:

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

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 [7; 11].

Syntax. A *disjunctive rule* r is a formula $a_1 \ \texttt{v} \ \cdots \ \texttt{v} \ a_n \ \texttt{:-} \ b_1, \cdots, b_k, \ \text{not}\ b_{k+1}, \cdots, \ \text{not}\ b_m.$ where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are atoms[5] and $n \geq 0$, $m \geq k \geq 0$. A literal is either an atom a or its default negation $\text{not}\ a$. Given a rule r, let $H(r) = \{a_1, ..., a_n\}$ denote the set of head literals, $B^+(r) = \{b_1, ..., b_k\}$ and $B^-(r) = \{\text{not}\ b_{k+1}, ..., \text{not}\ b_m\}$ the set of positive and negative body literals, resp., and $B(r) = B^+(r) \cup B^-(r)$. 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, $\text{not}.L = \{\text{not}.l \mid l \in L\}$.

Semantics. Here we briefly review the semantics of a disjunctive logic program, given by its stable models.

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 $B_{\mathcal{P}}$.[6] 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 $\mathcal{P}$.

---

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

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

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 $\mathcal{P}^X$.

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 $\mathcal{P}$, then all atoms in M are supported [17; 15].

An important property of stable models is related to the notion of *unfounded set* [21; 15]. 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[7]. Given $I \in \mathbf{I}_{\mathcal{P}}$, let $GUS_{\mathcal{P}}(I)$ (the *greatest unfounded set* of $\mathcal{P}$ w.r.t. I) denote the union of all unfounded sets for $\mathcal{P}$ w.r.t. I.

If M is a total interpretation for a program $\mathcal{P}$. M is a stable model of $\mathcal{P}$ iff $\text{not}.M = GUS_{\mathcal{P}}(I)$ [15].

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.

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 $DG_{\mathcal{P}}$ [3].

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 procedure in DLV for computing the deterministic consequences, namely DetCons. For the Model Generator, it is sufficient to say that it is similar to the Davis-Putnam procedure: It produces some "candidate" stable models I, each I is then verified by a function IsUnfoundedFree(I), which checks whether I is minimal. The Model Generator works on a ground instantiation $ground(\mathcal{P})$ of the logic program $\mathcal{P}$. More details can be found in [15; 9].

DetCons. 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 [4] 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"), here we treat "must be true" as true 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 [23] for better comparability: (i) Forward Inference, (ii) Kripke-Kleene Negation, (iii) Contraposition for True Heads, (iv) Contraposition for False Heads, (v) Well-founded Negation.

Rule (i) 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 (ii) derives an atom as false if no rule can support it. Rule (iii) applies if for a true atom only one rule is left that can support it, and makes inferences ensuring that this rule supports 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 (iv) 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 (v) sets all members of the greatest unfounded set to false. We note that rule (v) is only applied on recursive HCF subprograms for complexity reasons [4].

4 Backjumping

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 **:-** $b, c, \text{not } d.$ 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 only one way for a literal to become true unconditionally, i.e. no other literals entail

---

[7] 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 [15]).

its truth: These are the *chosen* literals. 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.

Determining Reasons for Derived Literals. The process of defining reasons for derived (non-chosen) literals is usually called *reason calculus*. For doing this, we have defined the auxiliary concepts of satisfying literals and orderings among satisfying literals for a given rule. Here, for lack of space, we do not report details for all rules that we have seen before. Details can be found in [20].

Reasons for Inconsistencies. We will now turn to how to exploit reasons during the computation. 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: (i) Deriving conflicting literals, and (ii) 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.

As mentioned above, inconsistencies from failing stability checks are a peculiarity of SDLP. This situation occurs if the function IsUnfoundedFree(I) returns false. Intuitively, this means that the current interpretation (which is guaranteed to be a model) is not stable.

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 reasons of literals which satisfy rules which contain unfounded atoms in their head (the cancelling assignments of these rules). Unsatisfied rules with unfounded atoms in their heads 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).

Using Inconsistency Reasons for Backjumping. When inside MG some inconsistency is detected (in DetCons or IsUnfoundedFree), we analyze the inconsistency reason, and can go directly to the highest 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 invocations 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 highest 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*.

Model Generator with Backjumping. In this section we describe MGBJ (shown in Fig. 1), a modification of the MG function, which is able to perform non-chronological backtracking.

It extends the “basic” model generator 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;
    $R(\mathrm{not}\ A)$ = { curr_level };
    if ( MGBJ ( $I \cup \{\mathrm{not}\ A\}$, 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;
};
```

Figure 1: 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 this 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. 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 $\mathrm{not}\ A$, 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 returns true also, while if it returned false, we check whether we backjump, setting IR and 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). We finally set bj_level to the maximum of the obtained inconsistency reason (or 0 if the set is empty) and return false.

5 Benchmarks

In order to evaluate the backjumping technique described in Section 4, we have implemented it as an extension of the DLV system. Judging from SAT, backjumping has the greatest impact on large, structured problem instances, so we have studied such instances. 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.

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 described backjumping technique.

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 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 [10]. 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 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, 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, 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.

Benchmark Problems 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 a rule which ensures that we either assume that variable x_i or its complement nx_i true: $x_i \,\mathrm{v}\, nx_i.$ 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.

We have randomly generated 20 3SAT instances for each problem size, in the hard region, by using a tool by Selman and Kautz, which is available at `ftp://ftp.research.att.com/dist/ai/` . 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` .

Moreover, to asses the impact of backjumping on Σ_2^P-complete problems we used "$\exists\forall$" Quantified Boolean Formulas (2QBF), which have already been used in the past for benchmarking SDLP systems [13; 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 [13], based on a reduction presented in [6].

The 2QBF formula Φ is valid iff the related propositional DLP $\mathcal{P}_\Phi$ has an answer set [6].

We used the benchmark instances from [13]. There, 50 hard instances per problem size were randomly generated. 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`.

6 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). We focus on structured SAT and

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

2-QBF instances. For random 3SAT, it is important to know that there is no or very small overhead when using backjumping.

Results without Lookahead We start with 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. 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 was quite impressive (about 80 millions).

About 2QBF, in Fig. 2, 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.

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.

Looking at the results of the structured SAT instances in Table 1, 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.

Finally, we report on the experiments with 2QBF instances. Fig. 3 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.

7 Conclusion

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 [23; 22], 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. 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. 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 respect to the heuristic method, and in particular, cooperates 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.

REFERENCES

[1] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2002.

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

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

[4] Francesco Calimeri, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Pruning Operators for Answer Set Programming Systems. In *Proceedings of the 9th International Workshop on Non-Monotonic Reasoning (NMR'2002)*, pages 200–209, April 2002.

[5] Rina Dechter and Daniel Frost. Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence*, 136(2):147–188, 2002.

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

[7] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative Problem-Solving Using the DLV System. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000.

[8] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.

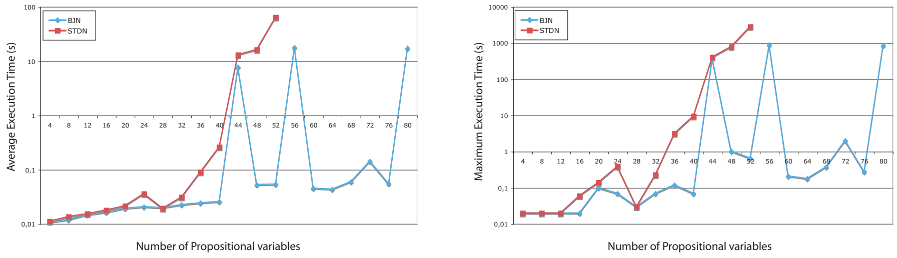

Figure 2: Execution time on random 2QBF problems without lookahead

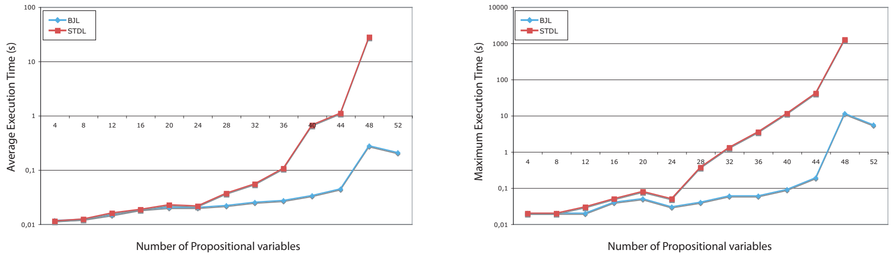

Figure 3: Execution time on random 2QBF problems with lookahead

[9] Wolfgang Faber. *Enhancing Efficiency and Expressiveness in Answer Set Programming Systems*. PhD thesis, Institut für Informationssysteme, Technische Universität Wien, 2002.

[10] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Experimenting with Heuristics for Answer Set Programming. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI) 2001*, pages 635–640, Seattle, WA, USA, August 2001. Morgan Kaufmann Publishers.

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

[12] Christoph Koch, Nicola Leone, and Gerald Pfeifer. Enhancing Disjunctive Logic Programming Systems by SAT Checkers. *Artificial Intelligence*, 15(1–2):177–212, December 2003.

[13] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 2005. To appear. Available via `http://www.arxiv.org/ps/cs.AI/0211004`.

[14] Nicola Leone, Riccardo Rosati, and Francesco Scarcello. Enhancing Answer Set Planning. In Alessandro Cimatti, Héctor Geffner, Enrico Giunchiglia, and Jussi Rintanen, editors, *IJCAI-01 Workshop on Planning under Uncertainty and Incomplete Information*, pages 33–42, August 2001.

[15] Nicola Leone, Pasquale Rullo, and Francesco Scarcello. Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation. *Information and Computation*, 135(2):69–112, June 1997.

[16] Vladimir Lifschitz. Answer Set Planning. In Danny De Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, pages 23–37, Las Cruces, New Mexico, USA, November 1999. The MIT Press.

[17] W. Marek and V.S. Subrahmanian. The Relationship between Logic Program Semantics and Non-Monotonic Reasoning. In *Proceedings of the 6th International Conference on Logic Programming – ICLP'89*, pages 600–617. MIT Press, 1989.

[18] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, June 2001.

[19] Patrick Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9:268–299, 1993.

[20] Francesco Ricca, Wolfgang Faber, and Nicola Leone. A backjumping technique for disjunctive logic programming. *AI Communications, to appear*, 2006.

[21] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.

[22] Jeffrey Ward. *Answer Set Programming with Clause Learning*. PhD thesis, Ohio State University, Cincinnati, Ohio, USA, 2004.

[23] Jeffrey Ward and John S. Schlipf. Answer Set Programming with Clause Learning. In Vladimir Lifschitz and Ilkka Niemelä, editors, *Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-7)*, LNCS, pages 302–313. Springer, January 2004.