# Solving Hard Disjunctive Logic Programs Faster (Sometimes)

Gerald Pfeifer

Institut f. Informationssysteme, TU Wien
Favoritenstr. 9-11, A-1040 Wien, Austria
gerald@pfeifer.com

**Abstract.** Disjunctive Logic Programming (DLP) under the consistent answer set semantics is an advanced formalism for knowledge representation and reasoning. It is, under widely believed assumptions, strictly more expressive than normal (disjunction-free) logic programming, whose expressiveness is limited to properties decidable in NP.

However, this higher expressiveness comes at a computational cost, and while there are now several efficient systems for normal logic programming under the answer set semantics, we are only aware of two serious implementations for the full, disjunctive case.

In this paper we investigate a novel technique to couple two main modules usually employed for the implementation of a DLP system more tightly: a model generator (which generates model candidates using a backtracking procedure) and a model checker (which verifies whether such a candidate indeed is an answer set). Instead of using the model checker only as a boolean oracle, in our approach, for every failed check, the model checker also returns a so-called unfounded set. Intuitively, this set provides a diagnosis why the model candidate is not an answer set, and the generator employs this knowledge to backtrack until the set is no longer unfounded, which is vastly more efficient than employing full-fledged model checks to control backtracking.

We implemented this approach in DLV, the leading implementation of DLP according to recent comparisons, and experiments on hard benchmark instances indeed show a significant speedup.

## 1 Introduction

Disjunctive Logic Programming (DLP) without function symbols under the consistent answer set semantics [GL91], also called Answer Set Programming, has slowly but steadily gained popularity since its inceptions in the early nineties of the last century and now serves as an advanced formalism for knowledge representation and reasoning in areas such as planning [DNK97,EFL+03,DKN02], software configuration, model checking, and advanced deductive database applications that involve complex knowledge manipulations on large databases at CERN; see [JNS+03,LPF+02] for further references.

There are a number of efficient implementations for the normal (non-disjunctive) case, which include Smodels [Sim96,SNS02], DLV [FP96,LPF+02], and

ASSAT [Zha02,LZ02] as well as Cmodels which is applicable on a subset of the language, so-called tight programs [Bab02].

In the disjunctive case, on the other hand, DLV used to be the only serious system (apart from proof-of-concept research prototypes) until GnT arrived at the scene [JNSY00], though recent studies still indicate DLV having an edge performance-wise [JNS+03,LPF+02].

Originally, few applications really required the higher expressivity of DLP which allows to express every property of finite structures decidable in the complexity class $\Sigma_2^P$ ($=$ $\mathrm{NP}^{\mathrm{NP}}$, versus $\mathrm{NP}$ for normal logic programming, which means that under widely believed assumptions DLP is strictly more expressive [EGM97]).

However, the use of DLP has been changing recently: First, disjunction also allows for a more natural representation of problems not requiring the higher expressivity (and in fact DLV, for example, detects such cases and avoids the overhead required for harder instances). And second, several applications from domains such as planning have been suggested and are under implementation [EFL+03,LRS01], which do require the full expressive power of DLP; having an efficient implementation of the full language is paramount for these.

In this paper, we provide a description of the intricate interaction of two main modules of DLV, the model generator and the model checker, and we describe novel optimization techniques related to this interaction that we implemented as part of the latest release of DLV.

Usually, for example in the implementations of DLV and GnT, a model generator incrementally constructs model candidates using a backtracking procedure, and a model checker then verifies whether these candidates indeed are answer sets.

One optimization, implemented in GnT and DLV and first described in [JNSY00], is to perform partial model checks after a failed regular model check and backtrack until the (increasingly smaller) partial interpretation passes such a partial check. In this paper we explore the possibility to use the model checker not just as a boolean oracle, but also let it return a so-called unfounded set. Intuitively, this set provides a diagnosis why the model candidate is not an answer set, and the generator can employ this knowledge during backtracking to avoid the more costly full partial model checks mentioned above in many cases.

We implemented and refined this novel approach, and indeed our experimentation on hard instances of QBF, a $\Sigma_2^P$-complete problem, shows a very nice speed-up.

## 2 Disjunctive Logic Programming

In this section we briefly introduce (function-free) Disjunctive Logic Programming (DLP) under the consistent answer set semantics, provide a high-level overview of the implementation of DLV, and finally review previous results on model checking. For further background we refer to [GL91,EGM97,Bar02,LPF+02].

### 2.1 Syntax

A variable or constant is a *term*. An *atom* is of the form $p(t_1, ..., t_n)$, where $p$ is a *predicate* of arity $n \geq 0$ and $t_1, ..., t_n$ are terms. A *classical literal* is an atom $a$ or a classically negated atom $\neg a$. A negation as failure literal (short *literal*) is either a positive literal $c$ or a negative literal $\text{not } c$, where $c$ is a classical literal.

A *(disjunctive) rule* $r$ is a clause of the form

$$a_1 \vee \cdots \vee a_n \leftarrow b_1 \wedge \cdots \wedge b_k \wedge \text{not } b_{k+1} \wedge \cdots \wedge \text{not } b_m. \qquad n \geq 1,\ m \geq 0$$

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are classical literals and $r$ needs to be *safe*, i.e., each variable occurring in $r$ must appear in one of the positive body literals $b_1, \cdots, b_k$ as well. The disjunction $a_1 \vee \cdots \vee a_n$ is the *head* of $r$, while the conjunction $b_1 \wedge \ldots \wedge b_k \wedge \text{not } b_{k+1} \wedge \ldots \wedge \text{not } b_m$ is the *body* of $r$.

We denote by $H(r)$ the set $\{a_1, \ldots, a_n\}$ of the head literals, and by $B(r)$ the set $\{b_1, \ldots, b_k, \text{not } b_{k+1}, \ldots, \text{not } b_m\}$ of the body literals. $B^+(r)$ (resp., $B^-(r)$) denotes the set of classical literals occurring positively (resp., negatively) in $B(r)$: $B^+(r) = \{b_1, \ldots, b_k\}$ and $B^-(r) = \{b_{k+1}, \ldots, b_m\}$.

*Constraints* are rules with an empty head ($n = 0$) which we use as syntactic sugaring equivalent to a rule $f \leftarrow b_1 \wedge \cdots \wedge b_k \wedge \text{not } b_{k+1} \wedge \cdots \wedge \text{not } b_m \wedge \text{not } f$ for some new propositional (i.e., nullary) atom $f$.

A *program* is a finite set of rules and constraints. A $\text{not}$-free (resp., $\vee$-free) program is called *positive* (resp., *normal*). An atom, a literal, a rule, a constraint, or a program, resp., is *ground* if it does not contain any variables.

A finite ground program is also called *propositional*, and in the rest of this paper we will focus on such programs, for the process of computing the ground equivalent of a program with variables is orthogonal to the issues at hand and has been the topic of separate research, see [SNS02,Syr02,ELM+98], for example.

### 2.2 Semantics

Before we can introduce the answer set semantics (also called stable model semantics) for disjunctive logic programs, we need a few prerequisites.

For any program $\mathcal{P}$, let the *Herbrand Universe* $U_{\mathcal{P}}$ be the set of all constants appearing in $\mathcal{P}$. In case no such constant exists, an arbitrary constant $\psi$ is added to $U_{\mathcal{P}}$. Furthermore, let the *Herbrand Literal Base* $B_{\mathcal{P}}$ be the set of all ground (classical) literals constructible from the predicate symbols appearing in $\mathcal{P}$ and the constants of $U_{\mathcal{P}}$.

For any rule $r$, the Ground Instantiation $Ground(r)$ denotes the set of rules obtained by applying all possible substitutions $\sigma$ from the variables in $r$ to elements of $U_{\mathcal{P}}$. For any program $\mathcal{P}$, $Ground(\mathcal{P})$ denotes the set $\bigcup_{r \in \mathcal{P}} Ground(r)$. For propositional programs, we trivially have that $\mathcal{P} = Ground(\mathcal{P})$ holds.

Following [Lif96], we define the *Answer Sets* of a program $\mathcal{P}$ in two steps, using the ground instantiation $Ground(\mathcal{P})$: first we define the answer sets of positive programs; then we give a reduction of programs containing negation as failure to positive ones and use that to define answer sets of arbitrary programs.

Step 1: A *(total) interpretation* $I$ is a set of ground classical literals, i.e., $I \subseteq B_{\mathcal{P}}$ w.r.t. a program $\mathcal{P}$. A consistent interpretation $X \subseteq B_{\mathcal{P}}$ is called *closed under a positive program* $\mathcal{P}$, if, for every $r \in Ground(\mathcal{P})$, $H(r) \cap X \neq \emptyset$ whenever $B(r) \subseteq X$. An interpretation $X$ is an *answer set* for a positive program $\mathcal{P}$, if it is minimal (under set inclusion) among all interpretations that are closed under $\mathcal{P}$.[1]

*Example 1. The positive program* $\mathcal{P}_1 = \{a \vee \neg b \vee c.\}$ *has the answer sets* $\{a\}$, $\{\neg b\}$, *and* $\{c\}$. *Its extension* $\mathcal{P}_2 = \{a \vee \neg b \vee c., \leftarrow a.\}$ *has the answer sets* $\{\neg b\}$ *and* $\{c\}$. *Finally,* $\mathcal{P}_3 = \mathcal{P}_2 \cup \{\neg b \leftarrow c.,\ c \leftarrow \neg b.\}$ *has the single answer set* $\{\neg b, c\}$. □

Step 2: The *reduct* or *Gelfond-Lifschitz transform* of a ground program $\mathcal{P}$ w.r.t. a set $X \subseteq B_{\mathcal{P}}$ is the positive ground program $\mathcal{P}^X$, obtained from $\mathcal{P}$ by

1. deleting all rules $r \in \mathcal{P}$ for which $B^-(r) \cap X \neq \emptyset$ holds;
2. deleting the negative body from the remaining rules.

Finally, an answer set of a (non-ground) program $\mathcal{P}$ is a set $X \subseteq B_{\mathcal{P}}$ such that $X$ is an answer set of $Ground(\mathcal{P})^X$.

*Example 2. Given the general program* $\mathcal{P}_4 = \{a \vee \neg b \leftarrow c.,\ \neg b \leftarrow \text{not } a, \text{not } c.,\ a \vee c \leftarrow \text{not } \neg b.\}$ *and* $I = \{\neg b\}$, *the reduct* $\mathcal{P}_4^I$ *is* $\{a \vee \neg b \leftarrow c.,\ \neg b.\}$. *It is easy to see that* $I$ *is an answer set of* $\mathcal{P}_4^I$, *and thus it is also an answer set of* $\mathcal{P}_4$.

*Now consider* $J = \{a\}$. *The reduct* $\mathcal{P}_4^J$ *is* $\{a \vee \neg b \leftarrow c.,\ a \vee c.\}$, *and we can easily verify that* $J$ *is an answer set of* $\mathcal{P}_4^J$, *so it is also an answer set of* $\mathcal{P}_4$.

*If, on the other hand, we take* $K = \{c\}$, *the reduct* $\mathcal{P}_4^K$ *is equal to* $\mathcal{P}_4^J$, *but* $K$ *is not an answer set of* $\mathcal{P}_4^K$: *for* $r = a \vee \neg b \leftarrow c$, *the condition* $B(r) \subseteq K$ *holds, but* $H(r) \cap K \neq \emptyset$ *does not. Indeed,* $I$ *and* $J$ *are the only answer sets of* $\mathcal{P}_4$. □

### 2.3 Answer Set Computation

In Figure 1 we provide a high-level description of the backtracking model generating procedure of the DLV system, which is similar to the one of GnT and the Davis-Putnam procedures commonly employed by SAT solvers [DP60].

For simplicity, this description assumes that the program $\mathcal{P}$ as well as auxiliary data structures are globally accessible, and it omits the processes of parsing, computing a suitable ground version of the (possibly) non-ground input, and output.

The computation is started by invoking ModelGenerator() with the empty three-valued interpretation where every classical literal in $B_{\mathcal{P}}$ is set to undefined.[2] If $\mathcal{P}$ has an answer set which is a superset of $I$, ModelGenerator() returns true and sets $I$ to this answer set; it returns false otherwise.

First the function DetCons() computes the deterministic consequences derivable from $\mathcal{P}$ and $I$; it returns false if this results in inconsistency, in which case also ModelGenerator() backtracks and returns false. If no inconsistency occurred, and no literal in $I$ is left undefined, we have found a model candidate and invoke the model checker to

[1] Note that we only consider *consistent answer sets*, while in [GL91] also the inconsistent set of all possible literals can be a valid answer set.

[2] In case of a three-valued interpretation every classical literal in $B_{\mathcal{P}}$ is either true, false, or undefined. If a literal assumes more than one of these truth values, the interpretation is inconsistent.

```
function ModelGenerator(var I : 3-Valued-Interpretation) : bool;
begin
    if not DetCons(I) then return false;
    if “no atom is undefined in I” then
        return IsAnswerSet(I);
    Select an undefined atom A using heuristics;
    if ModelGenerator(I ∪ {A}) then
        return true;
    else
        return ModelGenerator(I ∪ {not A});
end function ;
```

**Fig. 1.** Answer Set Computation

determine whether this is indeed an answer set. Else we choose one of the undefined literals, assume it true and recurse; in case this does not lead to an answer set, we assume the complement of that literal true (that is, we assume the literal itself false) and recurse as well. This proceeds until we either encounter an answer set or we have exhausted the entire search space.

We can easily see that there are three sources of complexity here in addition to the backtracking search itself: DetCons(), choosing which undefined atom to select, and the model check performed by IsAnswerSet().

By means of suitable data structures based on work by Dowling and Gallier [DG84], DLV performs DetCons() in linear time [CFLP02], so the heuristics which select an undefined atom $A$ and the implementation of IsAnswerSet() remain paramount for performance. [SNS02] and [FLP01] provide more details on heuristics for the normal and disjunctive cases, respectively, and [KLP03] provides an in-depth description of the model checker of the DLV system.

### 2.4 Model Checking

In the following we review some previous results on model checking [KLP03,LRS97] and then proceed with improving upon the basic algorithm described in Section 2.3.

The crucial concept for model checking in DLV is the notion of unfounded sets, which better lends itself for implementation than the original definition of answer sets.

**Definition 1.** (based on Definition 3.1 in [LRS97] and [KLP03]) Let $I$ be a total interpretation for a program $\mathcal{P}$. A set $X \subseteq B_{\mathcal{P}}$ of ground classical literals is an *unfounded set* for $\mathcal{P}$ w.r.t. $I$ if, for each rule $r \in Ground(\mathcal{P})$ such that $X \cap H(r) \neq \emptyset$, at least one of the following conditions holds:

$C_1$. $(B^+(r) \not\subseteq I) \vee (B^-(r) \cap I \neq \emptyset)$, that is, the body is false w.r.t. $I$.
$C_2$. $B^+(r) \cap X \neq \emptyset$, that is, some positive body literal belongs to $X$.
$C_3$. $(H(r) - X) \cap I \neq \emptyset$, that is, an atom in the head, distinct from the elements in $X$, is true w.r.t. $I$.

If $I$ is a partial interpretations, we first remove all literals which are undefined in $I$ from $\mathcal{P}$, and then proceed analogously to the total case above.

An interpretation $I$ for a program $\mathcal{P}$ is called *unfounded-free* if and only if no non-empty subset of $I$ is an unfounded set for $\mathcal{P}$ w.r.t. $I$. $\Box$

Intuitively, the presence of an unfounded set $X \subseteq I$ w.r.t. a model $I$ of $\mathcal{P}$ indicates that $I$ is not an answer set, because it is not minimal and some of its elements can be removed such that $I$ still remains a model. Formally, this can be stated as follows:

**Proposition 1.** *(Theorem 4.6 in [LRS97]) Let $I$ be a model for a program $\mathcal{P}$. $I$ is an answer set of $\mathcal{P}$ if and only if it is unfounded-free.* $\Box$

*Example 3. Consider $\mathcal{P}_2$ from Example 1, which has three models: $\{\neg b\}$, $\{c\}$, and $\{\neg b, c\}$. The first two are trivially unfounded-free, for they do not have any non-empty proper subset and are not unfounded sets themselves, so both are in fact answer sets. $\{\neg b, c\}$, on the other hand, contains two unfounded sets, namely $\{\neg b\}$ and $\{c\}$, and is therefore not an answer set.*

In general, checking whether a model $I$ for $\mathcal{P}$ is an answer set is a $\mathrm{co}$-$\mathrm{NP}$-complete task, and DLV solves it by means of a translation of $\mathcal{P}$ and $I$ to a satisfiability (SAT) problem which is unsatisfiable if and only if $I$ is unfounded-free (and therefore an answer set). If we consider the resulting SAT instance not as a decision problem, but as a functional problem, its solutions are the unfounded sets for $\mathcal{P}$ w.r.t. $I$. For further details we refer to [KLP03].

## 3 Model Generation and Checking Interplay

As mentioned before, both GnT and DLV implement an optimization first described in [JNSY00], where once a (total) model check fails, we backtrack and perform partial model checks during backtracking until we reach a partial model (a 3-valued interpretation) which passes such a partial check, or the root of the search tree.

To that end, we add a new global flag which, when set to "check_failed", indicates that we are in this special backtracking mode. And we add a function IsAnswerSet_Partial() which is similar to IsAnswerSet(), but ignores undefined literals occurring in rules (and constraints). That way it returns true if and only if $I$ contains an unfounded set that will remain unfounded for *every possible totalization* of $I$, which allows us to continue backtracking, confident that there cannot be any solution left in that part of the search tree.

The full, updated algorithm is depicted in Figure 2.

We improve upon this algorithm by further exploiting the results from Section 2.4 and a simple, but momentous, observation: when performing partial model checks during backtracking, the unfounded set internally computed by IsAnswerSet_Partial() will often be the same as the one originally found by IsAnswerSet().

Now we know that checking whether a set of classical literals $ufset$ is unfounded w.r.t. a program $\mathcal{P}$ and a (total or partial) interpretation $I$ can be done in linear time.[3]

[3] This directly follows from Definition 1 under the assumption that checking whether a literal is contained in $ufset$ and whether it is true in $I$ can be done in O(1), which is the case for DLV.

```
var state : { normal, check_failed } := normal;
function ModelGenerator(var I : 3-Valued-Interpretation) : bool;
begin
    if not DetCons(I) then return false;
    if "no atom is undefined in I" then
        if not IsAnswerSet(I) then
            state:=check_failed; return false;
        else
            return true;
    Select an undefined atom A using heuristics;
    if ModelGenerator(I ∪ {A}) then
        return true;
    else
        if state = check_failed then
            if not IsAnswerSet_Partial(I) then
                return false;
            else
                state:=normal;
        return ModelGenerator(I ∪ {not A});
end function ;
```

**Fig. 2.** Employing Partial Model Checking

So, instead of using the model checker only as a boolean oracle, whenever it encounters an unfounded set we also have it extract and return that set. For successive partial model checks we then first test whether the set is still unfounded w.r.t. $I$. If this is the case, we know that also a full partial model check of $I$ would fail, avoid the costly full partial model check, and continue backtracking.

Otherwise, we need to bite the bullet and perform a full partial model check, as $I$ may nevertheless contain an unfounded set different from the original one (e.g., a subset of the latter). Fortunately, IsUnfoundedSet() is extremely light – indeed we sometimes dub this optimization "quick partial model checking" – and several experiments confirmed that even in cases where this optimization does not succeed very often (or not at all) the overhead is hardly measurable.

The full algorithm exploiting this new approach is displayed in Figure 3.

Finally, we further improve on this algorithm by also extending IsAnswerSet_Partial() as described above, and let it extract an unfounded set whenever it encounters a model which is not an answer set. That way, after one or more full partial model checks during backtracking, we may again switch into "quick mode" and a sequence of expensive full partial model checks can well be decomposed into alternating sequences of full and quick partial checks.

In terms of pseudo-code, we only need to replace the following two lines in Figure 3

```
known_uff := not IsAnswerSet_Partial(I);
ufset:= ∅;
```

by

```
known_uff := not IsAnswerSet_Partial(I,ufset);
```

```
 var state : { normal, check_failed } := normal;
     ufset: SetOfClassicalLiterals;
function ModelGenerator(var I : 3-Valued-Interpretation) : bool;
begin
    if not DetCons(I) then return false;
    if "no atom is undefined in I" then
        if not IsAnswerSet(I,ufset) then
            state:=check_failed; return false;
        else
            return true;
    Select an undefined atom A using heuristics;
    if ModelGenerator(I ∪ {A}) then
        return true;
    else
        if state = check_failed then
            var known_uff : bool := false;
            if ufset ≠ ∅ then
                known_uff := IsUnfoundedSet(ufset,I);
            if not known_uff then
                known_uff := not IsAnswerSet_Partial(I);
                ufset:= ∅;
            if known_uff then
                return false;
            else
                state:=normal;
        return ModelGenerator(I ∪ {not A});
end function ;
```

**Fig. 3.** Employing Optimized Partial Model Checking

## 4 Benchmarks

To assess the impact of the optimizations presented in Section 3, we use Quantified Boolean Formulas (2QBF), a well-known $\Sigma_2^P$-complete problem [Pap94] that already proved to be a suitable benchmark problem in other recent comparisons [KLP03,LPF$^+$02].

Given a Quantified Boolean Formula $\Phi = \exists X \forall Y \phi$, where $X$ and $Y$ are disjoint sets of propositional variables and $\phi = C_1 \vee \ldots \vee C_k$ is a formula in 3DNF[4] over $X \cup Y$, the problem is to decide whether $\Phi$ is valid or not.

The transformation from 2QBF to disjunctive logic programming is a variant of a reduction used in [EG95], where we separate the actual problem instances and the following general encoding $\mathcal{P}_{2QBF}$:

$$t(X) \vee f(X) \leftarrow exists(X).$$
$$t(Y) \vee f(Y) \leftarrow forall(Y).$$
$$w \leftarrow conjunct(X, Y, Z, Na, Nb, Nc) \wedge$$

[4] Disjunctive normal form with three propositional variables per clause.

$$
\begin{aligned}
& t(X) \wedge t(Y) \wedge t(Z) \wedge f(Na) \wedge f(Nb) \wedge f(Nc). \\
t(X) &\leftarrow w \wedge forall(X). \\
f(X) &\leftarrow w \wedge forall(X). \\
&\leftarrow \text{not } w. \qquad t(true). \qquad f(false).
\end{aligned}
$$

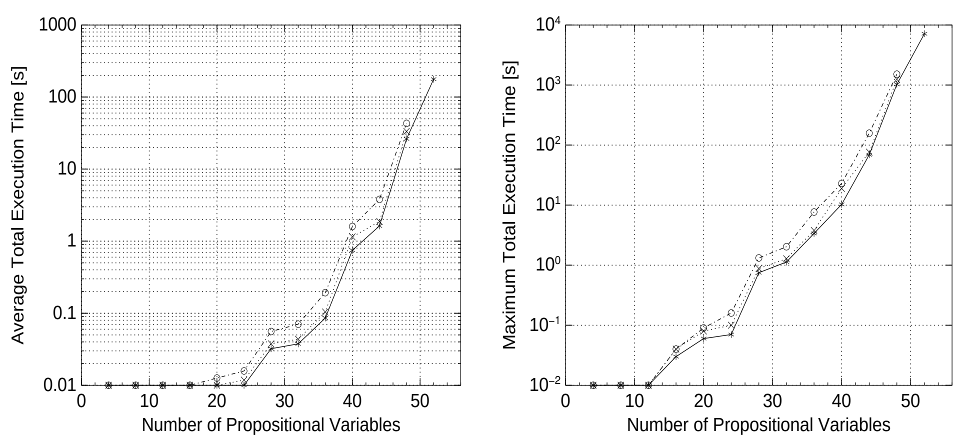
A concrete 2QBF instance $\Phi$ is then encoded by a set $F_\Phi$ of facts:

- $exists(v)$, for each existential variable $v \in X$;
- $forall(v)$, for each universal variable $v \in Y$; and
- $conjunct(p_1, p_2, p_3, q_1, q_2, q_3)$, for each disjunct $l_1 \wedge l_2 \wedge l_3$ in $\phi$, 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, $conjunct(x_1, true, y_4, false, y_2, false)$, encodes $x_1 \wedge \neg y_2 \wedge y_4$.

$\Phi$ is valid, if and only if $\mathcal{P}_{2QBF} \cup F_\Phi$ has an answer set.

*Benchmark Instances* We randomly generated 50 instances per problem size such that the number of $\forall$-variables is equal to the number of $\exists$-variables (that is, $|X| = |Y|$), each conjunct contains at least two universal variables. and the number of clauses is equal to the number of variables (that is, $|X| + |Y|$).

*Compared Systems* We took the 2003-05-16 release of DLV (with only minor and unrelated differences), and created three variants thereof: DLV$_{orig}$, which implements the strategy of Figure 2, DLV$'$, which employs the optimization described in Figure 3, and DLV$''$ with the additional optimization to update the cached unfounded set during backtracking.




o. . .DLV$_{orig}$, x. . .DLV$'$, *. . .DLV$''$

**Fig. 4.** Benchmark results for QBF

*Environment and Execution* Benchmarks were performed on an AMD Athlon 1.2 GHz machine with 512 MB of memory, using FreeBSD 4.8 and GCC 2.95 with -O3 optimization to generate executables. We allowed a maximum running time of 7200 seconds (2 hours) per instance and a maximum memory usage of 256 MB.

*Results* Cumulated results are provided in Figure 4. The graph to the left shows the average computation time for each system over the 50 instances per problem size; the graph to the right shows the maximum time taken. The plot for a system stops whenever that system failed to solve some problem instance within the given time and memory limits, and we can see that DLV$''$ was the only system able to solve all instances of size 52.

To study the performance of our optimizations in more detail (and also because the aggregate graphs are somewhat dominated by the relatively large number of simpler instances, cf. the differences between average and maximum times), we extracted the 34 hardest instances from our testbed. This includes several instances of larger sizes than were relevant for the full tests, where a system was killed once it encountered the first untractable instance.

Table 5 shows detailed results for those hard instances: specifically, overall execution time, total number of partial model checks, and number as well as percentage of quick partial model checks. (We omit the latter two for DLV$_{orig}$ where they are always zero by definition.)

## 5 Conclusions

Our benchmarks show that the optimizations we derived and implemented are a clear win, especially on hard instances of QBF where a significant amount of time is spent on (partial) model checking. Already DLV$'$ is a measurable improvement, but for DLV$''$, on average more than 50% of these partial checks enjoy the superior performance of the quick model checks, resulting in overall speedups of a factor of 2–3 in most cases.

These results are very encouraging and we plan to perform more extensive benchmarks, for example using encodings from planing domains, plus we are working to further improve internal data structures and algorithms related to the interplay of generator and checker. We have also tried to speculatively perform partial model checks while moving forwards (as opposed to backtracking) in the search tree and obtained very mixed results. Still, this is certainly an area worth of further investigations and we plan to revisit this issue.

*Acknowledgments* I am very grateful to Nicola Leone for fruitful discussions and suggestions related to this work as well as our long running cooperation in general.

Also, I would like to thank the other members of the DLV team whose continuous support provided the foundation to base this work on, especially Wolfgang Faber and Simona Perri as well as Tina Dell'Armi, Giuseppe Ielpa and Francesco Calimeri for their work on the core system and Christoph Koch. Thanks, finally, to the anonymous reviewers for valuable comments.

This work was supported by the European Commission under projects IST-2002-33570 INFOMIX, IST-2001-32429 ICONS, and IST-2001-37004 WASP.

| | Overall Execution | | | | Quick Partial Checks | | | |
|---|---|---|---|---|---|---|---|---|
| | Time [s] | | | Partial | Number | | Percentage | |
| Instance | $\mathrm{DLV}_{orig}$ | $\mathrm{DLV}'$ | $\mathrm{DLV}''$ | Checks | $\mathrm{DLV}'$ | $\mathrm{DLV}''$ | $\mathrm{DLV}'$ | $\mathrm{DLV}''$ |
| QBF 40.6 | 21.41 | 11.65 | 10.15 | 27493 | 15356 | 16821 | 55.9% | 61.2% |
| QBF 40.28 | 23.09 | 18.97 | 10.31 | 33663 | 2960 | 22000 | 8.8% | 65.4% |
| QBF 40.29 | 12.87 | 9.59 | 6.00 | 18206 | 4148 | 10955 | 22.8% | 60.2% |
| QBF 40.39 | 20.97 | 15.93 | 10.08 | 28041 | 5450 | 16140 | 19.4% | 57.6% |
| QBF 44.22 | 157.53 | 76.28 | 69.05 | 201807 | 132104 | 135038 | 65.5% | 66.9% |
| QBF 44.42 | 13.72 | 5.61 | 4.26 | 19314 | 14122 | 16444 | 73.1% | 85.1% |
| QBF 48.10 | 1517.73 | 1244.86 | 1026.35 | 1666943 | 210568 | 422252 | 12.6% | 25.3% |
| QBF 48.15 | 30.85 | 19.47 | 11.45 | 37743 | 14888 | 28432 | 39.4% | 75.3% |
| QBF 48.18 | 154.79 | 111.55 | 78.85 | 163636 | 44120 | 85724 | 27.0% | 52.4% |
| QBF 48.44 | 154.72 | 86.04 | 64.88 | 182439 | 86715 | 117204 | 47.5% | 64.2% |
| QBF 48.46 | 15.95 | 12.79 | 6.74 | 19020 | 2986 | 12300 | 15.7% | 64.7% |
| QBF 48.47 | 45.48 | 32.36 | 24.75 | 57605 | 15424 | 26380 | 26.8% | 45.8% |
| QBF 48.48 | 215.11 | 152.77 | 94.73 | 264435 | 70390 | 164814 | 26.6% | 62.3% |
| QBF 52.11 | 124.98 | 93.45 | 51.15 | 130943 | 29376 | 85568 | 22.4% | 65.3% |
| QBF 52.21 | 19.80 | 12.31 | 6.14 | 20695 | 8542 | 17354 | 41.3% | 83.9% |
| QBF 52.29 | 76.48 | 59.37 | 30.77 | 83294 | 16062 | 56013 | 19.3% | 67.2% |
| QBF 52.31 | 2812.77 | 1413.67 | 1232.85 | 3353087 | 1935988 | 2093868 | 57.7% | 62.4% |
| QBF 52.37 | 9.36 | 7.03 | 3.95 | 9800 | 2180 | 6148 | 22.2% | 62.7% |
| QBF 52.39 | 668.56 | 459.35 | 338.83 | 737280 | 209700 | 384930 | 28.4% | 52.2% |
| QBF 52.41 | – | – | 7183.19 | 8539647 | – | 1807494 | –% | 21.2% |
| QBF 56.23 | 692.54 | 291.47 | 205.93 | 694837 | 463000 | 578278 | 66.6% | 83.2% |
| QBF 60.11 | 785.75 | 426.62 | 333.51 | 729792 | 355689 | 461958 | 48.7% | 63.3% |
| QBF 60.20 | 2021.44 | 1649.66 | 730.83 | 1989945 | 199920 | 1446084 | 10.0% | 72.7% |
| QBF 60.21 | 19.30 | 9.83 | 6.60 | 18811 | 9846 | 14561 | 52.3% | 77.4% |
| QBF 60.31 | 40.62 | 26.06 | 12.27 | 40480 | 14694 | 33721 | 36.3% | 83.3% |
| QBF 60.33 | 265.13 | 184.51 | 105.12 | 241128 | 67753 | 162331 | 28.1% | 67.3% |
| QBF 60.36 | 55.33 | 43.85 | 18.79 | 50569 | 7366 | 38958 | 14.6% | 77.0% |
| QBF 64.31 | 59.52 | 37.31 | 18.59 | 56656 | 21868 | 46030 | 38.6% | 81.2% |
| QBF 64.33 | 740.64 | 420.30 | 370.40 | 637186 | 312417 | 333073 | 49.0% | 52.3% |
| QBF 64.34 | 29.09 | 20.92 | 9.56 | 24475 | 6176 | 18408 | 25.2% | 75.2% |
| QBF 64.16 | 2378.83 | 1778.20 | 945.31 | 2153919 | 505896 | 1490504 | 23.5% | 69.2% |
| QBF 64.6 | 4373.46 | 3010.92 | 1931.37 | 3156869 | 908358 | 1901420 | 28.8% | 60.2% |
| QBF 64.7 | 1027.07 | 875.09 | 311.22 | 723407 | 56068 | 566996 | 7.8% | 78.4% |
| QBF 76.39 | 341.25 | 276.19 | 119.20 | 226053 | 32172 | 159434 | 14.2% | 70.5% |

**Fig. 5.** Detailed benchmarks results for hardest QBF instances

## References


[Bab02] Y. Babovich. Cmodels homepage, since 2002. `http://www.cs.utexas.edu/users/tag/cmodels.html`.

[Bar02] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2002.

[CFLP02] F. Calimeri, W. Faber, N. Leone, and G. Pfeifer. Pruning Operators for Answer Set Programming Systems. In *NMR'2002*, pp. 200–209, April 2002.

[DG84] W. F. Dowling and J. H. Gallier. Linear-time Algorithms for Testing the Satisfability of Propositional Horn Formulae. *JLP*, 3:267–284, 1984.

[DKN02] J. Dix, Ugur Kuter, and D. Nau. Planning in Answer Set Programming using Ordered Task Decomposition. *TPLP*, October 2002. Revised version under submission. Short paper to appear in KI 2003 (German National Conference on Artificial Intelligence).

[DNK97] Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding Planning Problems in Nonmonotonic Logic Programs. In *ECP-97*, pp. 169–181. Springer, 1997.

[DP60] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *JACM*, 7:201–215, 1960.

[EFL+03] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A Logic Programming Approach to Knowledge-State Planning: Semantics and Complexity. *ACM TOCL*, 2003. To appear.

[EG95] T. Eiter and G. Gottlob. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *AMAI*, 15(3/4):289–323, 1995.

[EGM97] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *ACM TODS*, 22(3):364–418, September 1997.

[ELM+98] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. Progress Report on the Disjunctive Deductive Database System `dlv`. *FQAS'98*, pp. 148–163. Springer.

[FLP01] W. Faber, N. Leone, and G. Pfeifer. Experimenting with Heuristics for Answer Set Programming. In *IJCAI 2001*, pp. 635–640. Morgan Kaufmann Publishers.

[FP96] W. Faber and G. Pfeifer. DLV homepage, since 1996. `http://www.dlvsystem.com/`.

[GL91] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[JNS+03] T. Janhunen, I. Niemelä, D. Seipel, P. Simons, and Jia-Huai You. Unfolding Partiality and Disjunctions in Stable Model Semantics. Tech. Report cs.AI/0303009, arXiv.org, March 2003.

[JNSY00] T. Janhunen, I. Niemelä, P. Simons, and Jia-Huai You. Partiality and Disjunctions in Stable Model Semantics. *KR 2000, April 12-15*, pp. 411–419. Morgan Kaufmann.

[KLP03] C. Koch, N. Leone, and G. Pfeifer. Using SAT Checkers for Disjunctive Logic Programming Systems. *Artificial Intelligence*, 2003. To appear.

[Lif96] V. Lifschitz. Foundations of Logic Programming. *Principles of Knowledge Representation*, pp. 69–127. CSLI Publications, Stanford, 1996.

[LPF+02] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, C. Koch, C. Mateis, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. Tech. Report cs.AI/0211004, arXiv.org, November 2002. Submitted to ACM TOCL.

[LRS97] N. Leone, P. Rullo, and F. Scarcello. Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation. *Information and Computation*, 135(2):69–112, June 1997.

[LRS01] N. Leone, R. Rosati, and F. Scarcello. Enhancing Answer Set Planning. *IJCAI-01 Workshop on Planning under Uncertainty and Incomplete Information*, pp. 33–42.

[LZ02] F. Lin and Y. Zhao. ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In *AAAI-2002*, AAAI Press / MIT Press.

[Pap94] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[Sim96] P. Simons. Smodels Homepage, since 1996. `http://www.tcs.hut.fi/Software/smodels/`.

[SNS02] P. Simons, I. Niemelä, and T. Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138:181–234, June 2002.

[Syr02] T. Syrjänen. Lparse 1.0 User's Manual, 2002. `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`.

[Zha02] Y. Zhao. ASSAT homepage, since 2002. `http://assat.cs.ust.hk/`.