

Disjunctive Answer Set Programming via Satisfiability

Yuliya Lierler

Erlangen-Nürnberg Universität
yuliya.lierler@informatik.uni-erlangen.de

Abstract. Using SAT solvers as inference engines in answer set programming systems showed to be a promising approach in building efficient systems. Nowadays SAT based answer set programming systems successfully work with nondisjunctive programs. This paper proposes a way to use SAT solvers for finding answer sets for disjunctive logic programs. We implement two different ways of SAT solver invocation used in nondisjunctive answer set programming. The algorithms are based on the definition of completion for disjunctive programs and the extension of loop formula to the disjunctive case. We propose the necessary modifications to the algorithms known for nondisjunctive programs in order to adapt them to the disjunctive case and demonstrate their implementation based on system CMODELS.

1 Introduction

Disjunctive logic programming under the stable model semantics [GL91] is a new methodology called *answer set programming* (ASP) for solving combinatorial search problems. It is a form of declarative programming related to logic programming languages, such as Prolog. In answer set programming, solutions to a problem are represented by answer sets, and not by answer substitutions produced in response to a query as in conventional logic programming. Instead of Prolog systems, this programming method uses answer set solvers, such as SMOBELS [SS05], CMOBELS [Le05], ASSAT [LZ02], DLV [LPE⁺05], and GNT [Jea05]. These efficient systems made it possible for ASP to be successfully applied in such areas as planning, bounded model checking, historical linguistics and product configuration.

Systems DLV and GNT are more general as they work with the class of disjunctive logic programs, while other systems cover only nondisjunctive case. Both systems CMOBELS and ASSAT use SAT solvers as search engines. They are based on the relationship between the completion semantics [Cla78] and answer set semantics for logic programs. It is well known that all answer sets of a program are also models of its completion while the converse is not always true. For the big class of programs, called *tight*, for which the converse holds SAT solvers can serve a role of answer set enumerators. Lin and Zhao [LZ02] found a way to use SAT solvers for computing answer sets for also nontight nondisjunctive programs. Systems implementing the approach, ASSAT and CMOBELS, showed that SAT based answer set programming is promising by providing the successful experimental analysis with respect to the other state-of-the-art ASP systems [LZ02,LM04,GLM04]. At the same time CMOBELS proved to be an efficient system in such real-world applications as the wire-routing problem [EW04],

and the problem of reconstructing probable phylogenies in the area of historical linguistics [BEMR05].

This paper proposes the way to use SAT solvers also for finding answer sets for disjunctive logic programs. The work is based on the definition of completion for disjunctive programs [LL03] and the extension of loop formula definition [LZ02] to the case of disjunctive programs [LL03]. We propose necessary modifications to the SAT based ASSAT algorithm [LZ02] as well as to the generate and test algorithm from [GLM04] in order to adapt them to the case of disjunctive programs. We implement the algorithms in system CMODELS and demonstrate experimental results.

Existing systems DLV [LPE⁺05,KLP03] and GNT [Jea05] implement generate and test approach for finding answer sets for disjunctive programs. DLV implements a unique search algorithm for generating candidate models and uses SAT solver for testing them. GNT utilises answer set system for nondisjunctive programs SMODELS for both procedures: generating and testing. The difference in our work is that we propose use of SAT solver as an engine for these tasks: first generating candidate solutions, and second testing them. In our approach for the class of tight disjunctive programs testing part of the procedure is not necessary.

The paper is organised as follows. First we introduce terminology and theory needed to extend SAT based algorithms for nondisjunctive programs to the disjunctive case. We demonstrate the algorithms themselves, provide the details of program's syntax admitted by the implementation and show the preliminary experimental results. The last section contains proofs of the theoretical results.

2 Theoretical Preliminaries

Disjunctive program (DP) is a set of rules with expressions that have the form

$$A \leftarrow B, F \tag{1}$$

where A is the head of the rule and is a disjunction of atoms or symbol \perp , B is a conjunction of atoms, and F is a formula of the following form

$$\text{not } A_1, \dots, \text{not } A_m, \text{not not } A_{m+1}, \dots, \text{not not } A_n$$

We call such rules *disjunctive rules*.

If a head of a rule does not contain a disjunction, we call such rule *nondisjunctive*. If formula F of rule (1) contains an expression of the form *not not* A_i then the rule is called *nested*, otherwise the rule is *non-nested*. If all rules of a DP are nondisjunctive we call such program nondisjunctive as well.

Let Π be a DP whose rules have the form

$$A \leftarrow \text{Body}. \tag{2}$$

We identify a disjunction of atoms A with a set of atoms occurring in A . A completion of Π [LL03], $\text{Comp}(\Pi)$, is defined to be a set of propositional formulas that consists of the implication

$$\text{Body} \supset A \tag{3}$$

for every rule (2) in Π , and the implication

$$a \supset \bigvee_{A \leftarrow \text{Body} \in \Pi, a \in A} (\text{Body} \wedge \bigwedge_{p \in A \setminus \{a\}} \neg p) \quad (4)$$

for each atom $a \in \Pi$.

The definition of when a set of atoms satisfies a rule (or a head of a rule, or a body of a rule) is the usual definition of satisfaction in propositional logic, with the comma understood as conjunction, "not" as negation, and $A \leftarrow \text{Body}$ as the material implication $\text{Body} \supset A$. We say that set of atoms X *satisfies* program Π (symbolically, $X \models \Pi$) if X satisfies every rule of Π . We call such set of atoms X a model of program Π .

The reduct of rule (1)

$$A^X \leftarrow B^X, F^X$$

with respect to set of atoms X is defined as follows:

- $A^X = A$
- $B^X = B$
- $F^X = (\text{not } A_1)^X, \dots, (\text{not } A_m)^X, (\text{not not } A_{m+1})^X, \dots, (\text{not not } A_n)^X$ where $(\text{not } A_i)^X = \perp$ and $(\text{not not } A_i)^X = \top$ if $X \models A_i$ otherwise $(\text{not } A_i)^X = \top$ and $(\text{not not } A_i)^X = \perp$.

The *reduct* Π^X of program Π with respect to X is a set of rules $A^X \leftarrow B^X, F^X$ for all rules (1) in Π . Set X of atoms is an *answer set* [LL03] for a program Π if X is minimal among the sets of atoms that satisfy the reduct Π^X .

Let Π be a DP. A *positive dependency graph* of Π is directed graph G such that

- vertices of G are the atoms occurring in Π
- for every rule (1) in Π , G has an edge from each atom $a \in A$ to each atom in B .

A disjunctive program is *tight* [LL03] if its positive dependency graph is acyclic.

Theorem for Tight Programs. [LL03] *For any tight disjunctive program Π and any set X of atoms, X is an answer set for Π iff X satisfies $\text{comp}(\Pi)$.*

A nonempty set of atoms L is called a *loop* of Π if for any pair a_1, a_2 of atoms in L , there exists a path of nonzero length from a_1 to a_2 in the positive dependency graph of Π such that all vertices in this path belong to L . *Loop formula* F_L has the form

$$\bigwedge L \supset \bigvee R(L) \quad (5)$$

where $R(L)$ is a set of formulas

$$B \wedge F \wedge \bigwedge_{p \in A \setminus L} \neg p \quad (6)$$

for all rules (1) in Π such that $A \cap L \neq \emptyset$ and $B \cap L = \emptyset$ [LL03]. (By $\bigwedge L$ we denote the conjunction of all elements of L , and $\bigvee R(L)$ is understood in an analogous way.)

Theorem 1 (Theorem 1 in [LL03]). *For any DP Π and any set X of atoms, X is an answer set for Π iff X is a model of $\text{Comp}(\Pi) \cup \text{LF}(\Pi)$ where $\text{LF}(\Pi)$ stands for the set of all loop formulas for the program.*

3 Theoretical Basis for Modifications

Based on Theorem for Tight Programs for the large class of *tight* programs answer sets for a tight program are the same as the models of its completion, and hence SAT solvers can play the role of answer set enumerators. Checking the existence of an answer set for a tight disjunctive program forms an NP-complete decision problem as in the non-disjunctive case.

For the class of nontight nondisjunctive programs [LZ02] proposed a SAT based ASSAT algorithm that employed a loop formula concept.

ASSAT procedure [LZ02]:

- 1 Let T be the Completion of Π — $Comp(\Pi)$
- 2 Find a model M of T . If there is no such model then terminate with failure.
- 3 If M is an answer set, then exit with it.
- 4 Find a loop L , such that its loop formula F_L is not satisfied by the current model.
- 5 Let T be $T \cup F_L$ and go back to step 2.

In [GLM04] the authors proposed the modification to ASSAT procedure by allowing more flexible use of SAT solvers. The authors first considered standard SAT Davis-Logemann-Loveland (DLL) procedure [DLL62]. Once the completion of the program is calculated, DLL procedure is applied on it. DLL generates models of completion if applied with no changes. [GLM04] added *test* part into DLL procedure for verifying whether a generated model of the completion is an answer set. In case when *test* gives a negative result control is given back to DLL, and it proceeds with the search for the next model. The main advantage of this approach in comparison with ASSAT algorithm is avoiding the overhead of initialising the search tree of the SAT solver each time when *test* is performed with the negative result. We refer to this approach as generate and test algorithm.

With the definitions of completion and loop formula for disjunctive programs proposed in [LL03] we can adapt both former mentioned algorithms to a broader class of disjunctive programs.

In order to implement these procedures we first need to answer two questions:

- 1 How to verify if the model M is indeed an answer set?
- 2 How to find a loop formula which does not satisfy model M when M is not an answer set?

The answer to the first question lies in the minimality requirement of the definition of an answer set, i.e. set X of atoms is an answer set for a program Π if X is minimal among sets of atoms that satisfy the reduct Π^X . Let Π be a DP, and M be a set of atoms satisfying Π . It trivially follows from Lemma 3i [EL03] that M satisfies the reduct Π^M . Consider the formula F to be of the form $\Pi^M \cup M^- \cup \neg M$, where (i) by Π^M we denote the reduct of Π under M such that its rules $A \leftarrow Body$ are represented as material implication $\bigwedge Body \supset A$; (ii) M^- denotes the conjunction of negation of the atoms in Π that do not belong to M ; and (iii) by $\neg M$ we denote the negation of the conjunction of atoms in M . Based on the definition of an answer set, if formula F is

satisfied by some model M' (note that $M' \subset M$), then M is not an answer set of Π . We may now define *minimality test procedure* on program Π and model of its completion M . First, formula F is computed as $\Pi^M \cup M^- \cup \neg M$. Then SAT solver is invoked on clausified formula F . Last if F is unsatisfied then the verified model M is indeed an answer set of Π otherwise some model of F is returned. The minimality test procedure is similar to the one introduced in [JNSY00].

In described minimality test procedure, SAT solver is used for the model verification step. The idea of using SAT solvers for this task is introduced in [KLP03] where the concept of *unfounded-free* models of non-nested disjunctive programs is explored. This approach allows using some modularity property of the program, that permits splitting verification step on the whole program into verification on its parts. The algorithm in Figure 6 [KLP03] also makes use of the fact that minimality check can be performed in polynomial time for the class of *head cycle free* programs. Important future work is to explore how this approach can be extended to the case of nested disjunctive programs.

The answer to the second question "How to find a loop formula which does not satisfy model M when M is not an answer set?" lies in the following definitions and Proposition 1.

Definition 1. Let Π be a DP, and M a model of Π . We call some rule $A \leftarrow \text{Body} \in \Pi$ supporting atom a under set M if $A \cap M = \{a\}$, and $M \models \text{Body}$.

The definition below and proposition are closely related to Definition 3, and Theorem 2 given in [LZ02] for nondisjunctive programs.

Definition 2. Let Π be a DP, M a model of $\text{Comp}(\Pi)$, and M' a model of Π^M , such that $M' \subset M$. We say that a loop L of Π is a maximal loop under $M \setminus M'$ if L is a strongly connected component of $G_{M \setminus M'}$, where $G_{M \setminus M'}$ is a subgraph of the positive dependency graph of Π induced by $M \setminus M'$. A maximal loop L under $M \setminus M'$ is called a terminating one if there does not exist another maximal loop L_1 under $M \setminus M'$ such that for some $p \in L$ and $q \in L_1$, there is a path from p to q in $G_{M \setminus M'}$.

Proposition 1. Let Π be a DP, M a model of $\text{Comp}(\Pi)$, and M' a model of Π^M , such that $M' \subset M$. There must be a terminating loop of Π under $M \setminus M'$. Furthermore, M does not satisfy a loop formula of any of the terminating loops of Π under M .

Based on this proposition and already mentioned minimality test procedure we may outline the answer to the second question posed. Let Π be a DP and M a model of completion such that it is not an answer set of Π . In order to find a loop formula of Π unsatisfied by M , we first find model M' of formula $\Pi^M \cup M^- \cup \neg M$, and second look for a terminating loop of Π . Once such loop is found we compute its formula.

4 Details on the Modified Algorithms and the Implementation

Our implementation is enhanced to identify tightness feature of a disjunctive program. In case when a program is tight the system performs completion procedure on the program at first and uses SAT solver for enumerating its answer sets avoiding use of minimality test procedure on the models. This way we allow efficient use of SAT solvers

in ASP, by analysing program syntactically and identifying in advance disjunctive program involving lower computational complexity.

For the case of nontight disjunctive programs we base our modifications to ASSAT algorithm on the minimality test procedure and Proposition 1. Modified algorithm follows:

DP-assat-Proc

- 1 Let T be the Completion of Π — $Comp(\Pi)$
- 2 Invoke a SAT solver $SAT-A$ to find a model M of T . If there is no such model then terminate with failure.
- 3 Invoke a minimality test procedure on program Π , and model M with the SAT solver $SAT-B$ to find a model M' . If there is no such model then exit with an answer set M . If there is model M' then M is not an answer set of Π .
- 4 Build the subgraph $G_{M \setminus M'}$ of positive dependency graph of Π induced by $M \setminus M'$. Look for terminating loop L under $M \setminus M'$ in $G_{M \setminus M'}$.¹
- 5 Let T be $T \cup F_L$, where F_L is a loop formula of L and go back to step 2.

The algorithm utilises a SAT solver not only for finding models of completion but also for verifying whether a found model is an answer set. We expect the verification of model's minimality time not to exceed greatly the time of finding the model itself. First a set of clauses for this step is smaller, and second it is restricted by a model itself, in a sense that the negations of the atoms that do not belong to the model are added as unit clauses.

In the worst case *DP-assat-Proc* requires computing exponential number of loop formulas and hence brings the exponential grows to formula T passed to a SAT solver in Step 2. In following algorithm *DP-generate-test-enhanced-Proc* we address this problem by controlling the grows of formula T . There we add to the completion only one clause implied by a computed loop formula, instead of a loop formula itself.

Adapted generate and test algorithm from [GLM04] to the case of nontight disjunctive programs follows:

DP-generate-test-Proc

- 1 Compute completion of Π — $Comp(\Pi)$
- 2 Initiate the SAT solver $SAT-A$ with the completion $Comp(\Pi)$. Invoke DLL to find a model M of $Comp(\Pi)$. If there is no such model then terminate with failure.
- 3 The same as Step 3 of *DP-assat-proc*.
- 4 Return control to the $SAT-A$ procedure DLL for finding next model M of the completion. If there is no such model then terminate with failure. Go back to Step 3.

State-of-the-art SAT solvers are enhanced by the ability of performing not only simple backtracking within DLL procedure but also backjumping and learning if they are provided with a certain clause. Backjumping and learning techniques made it possible for

¹ Note that $G_{M \setminus M'}$ can be simplified by removing edges between the nodes of the graph that do not correspond to any rules in the reduct Π^M . The note is due to Liu Lengning and Mirek Truszczynski's observation.

the area of SAT solving to gain a great boost in the performance in the last decade. By supporting these features of modern SAT solvers we should gain the most pay-back in applying SAT based search in answer set programming. We analyse the notion of a loop formula and retrieve a necessary clause from it that allows us to enhance SAT solver inner computation.

DP-generate-test-enhanced-Proc

- 1-3 These steps are the same as Steps 1-3 in *DP-generate-test-Proc*
- 4 The same as step 4 in *DP-assat-proc*.
- 5 Calculate a clause Cl implied by F_L such that $M \not\models Cl$.
- 6 Return control to the *SAT-A* procedure DLL by giving Cl as a clause to backjump and learn. Find next model M of the completion. If there is no such model then terminate with failure. Go back to step 3.

Note that although in the worst case *DP-generate-test-enhanced-Proc* requires computing exponential number of loop formulas, a SAT solver may still work in polynomial space if it periodically deletes learned clauses. Within our implementation SAT solver SIMO performs this operation. Due to the fact that DLL procedure of SIMO at Step 6 continues to explore the same search tree we are guaranteed not to find the same models again.

It is worth to mention implementation details of the minimality test procedure, i.e. step 3 of the above algorithms. Let us first notice that for program Π and its model M , program's reduct Π^M is equivalent to $\Pi^+ \cup (\Pi \setminus \Pi^+)^M$ where by Π^+ we denote a part of the program whose rules (1) contain empty F . For programs whose Π^+ part is relatively large this observation may help to improve the performance of the minimality test procedure. SAT solver ZCHAFF permits in addition to storing permanent database of clauses, also adding and deleting clauses on demand. Thus we take clauses corresponding to Π^+ to be a permanent database of clauses for some program Π and at first invocation of ZCHAFF initialise it with Π^+ . Once we need to verify minimality of the model M we temporally add clauses $(\Pi \setminus \Pi^+)^M \cup M^- \cup \neg M$ to ZCHAFF database and search for models of such formula. Afterwards the temporally added clauses are deleted. By using these advanced features of SAT solver ZCHAFF we sometime may avoid repeating initialisation of large part of the formula corresponding to Π^+ needed for models' minimality verification.

5 Syntax of CMODELS

So far we presented the algorithms for the case of disjunctive programs that contain rules of the form (1). Our implementation – system CMODELS – uses the program LPARSE *--dlp-choice* for grounding disjunctive logic programs. The input of CMODELS may include rules of three types. It allows (i) non-nested disjunctive rules, (ii) choice rules that have the form

$$\{A_0, \dots, A_k\} \leftarrow A_1, \dots, A_l, \text{not } A_{l+1}, \dots, \text{not } A_m \quad (7)$$

```

% Sample graph encoding, i.e.,
% graph contains 3 nodes, and 3 edges:
% edges between nodes 1 and 2, 2 and 3, 3 and 1.
node(1..3). edge(1,2).edge(2,3).edge(3,1).
% Declaration of three colors
col(red). col(green). col(blue).
% Disjunctive rule: stating that node has some color
colored(X,red) | colored(X,green) | colored(X,blue) :- node(X).
% Neighboring nodes should not have the same color
:- edge(X,Y), colored(X,C), colored(Y,C), col(C).

```

Fig. 1. Encoding of 3-colorability problem for grounder LPARSE: *3-col.lp*

where A_i are atoms, and (iii) weight constraints of the form

$$A_0 \leftarrow L[A_1 = w_1, \dots, A_m = w_m, \text{not } A_{m+1} = w_{m+1}, \dots, \text{not } A_n = w_n] \quad (8)$$

where A_0 is an atom or the symbol \perp ; A_1, \dots, A_n are atoms; and L (lower bound), and $w_1 \dots w_n$ (weights) are integers.

The concept of an answer set for programs containing rules (7,8) was introduced in [NS00]. The original rules which are given to the front end LPARSE *--dlp-choice* are more expressive. They allow lower and upper bounds for choice rules and upper bounds for weight rules. They also allow use of literals in place of atoms. (A *literal* is a propositional atom possibly preceded by the classical negation symbol \neg .) LPARSE *--dlp-choice* uses auxiliary variables and translates all the rules to the forms specified above. Internally in CMODELS, choice rules are translated into nondisjunctive nested rules, while weight constraints are translated with the help of auxiliary variables [FL05].

Note that CMODELS is the first answer set programming system that allows use of disjunctive and choice rules simultaneously.

6 Experimental Analyses

First we provide the details on the performance of system CMODELS in the case of tight disjunctive programs [Lie05]. Such programs are of lower computational complexity than disjunctive programs in general, but nevertheless wide class of programs is tight and demonstrating these results is in favour of the described SAT based answer set programming approach.

Figure 1 presents the tight disjunctive program *3-col.lp* based on the encoding of 3-colorability problem provided at the DLV web site. Program *3-col.lp* can be written as the program with choice rule in place of disjunctive rule supported by systems SMOBELS, SMOBELScc [WS04] and CMODELS. This allows us to find answer sets of the program also by means of nondisjunctive answer set programming.

For experimental analyses we used the encoding of the 3-colorability problem as in Figure 1 on Simplex graph instances. We compared the performance of systems CMODELS, DLV, and GNT on disjunctive program and also SMOBELS, SMOBELScc and CMODELS on choice rule encoding of a problem. The experiments were run on Pentium 4, CPU 3.00GHz and presented in Figure 2.

simplex	lparse disj	lparse	cmodels mchaff disj	cmodels mchaff choice	cmodels zchaff disj	cmodels simo disj	smodelsCC choice	smodels choice	dlv.5 02.23 disj	gnt disj
30	0.04	0.05	0.09	0.12	0.05	0.06	0.07	0.31	0.63	1.47
60	0.19	0.24	0.31	0.39	0.25	0.28	0.37	7.85	9.72	26.73
120	0.84	1.08	1.22	1.54	0.97	1.54	1.47	138.48	196.36	442.31
240	3.38	4.16	5.22	6.24	4.09	13.63	6.30	-	-	-
360	7.56	10.60	11.44	14.45	9.16	52.36	14.52	-	-	-
480	13.69	17.25	20.11	25.97	17.16	175.08	27.99	-	-	-
600	21.58	27.33	35.32	47.02	27.04	369.50	48.03	-	-	-

Fig. 2. CMODELS, DLV, GNT on disjunctive programs versus CMODELS, SMODELScC and SMODELS on programs involving choice rules

The number in the first column characterising the instances stands for the number of levels in the simplex graph, respectively. The other columns represent the running times of the systems in seconds. "-" stands for the fact that the system was not able to conclude on the test problem within 30 minutes cutoff time.

The second column demonstrates that the ground disjunctive program is smaller than the corresponding ground program with the choice rules, due to more economical LPARSE encoding. CMODELS on disjunctive programs takes an advantage of a smaller ground program by producing fewer clauses and performing the search faster. CMODELS using SAT solver ZCHAFF² outperforms all other answer set programming systems. SMODELScC performance is comparable with the performance of CMODELS using MCHAFF⁷ on choice encoding. SMODELScC employs the similar heuristic in its search procedure as SAT solver MCHAFF. It is also worth to notice that CMODELS using SIMO³ is by the order of magnitude slower than CMODELS using ZCHAFF even though the underlying algorithms of both SAT solvers are similar. Capability of using different search engines may prove to be useful in practical applications.

For experimental analysis of the systems' performance on the nontight programs we shall specify the algorithmic differences of SAT solvers invocations. Algorithm *DP-assat-Proc* is implemented in CMODELS using SAT solver MCHAFF in Step 2 of the procedure. On the other hand algorithm *DP-generate-test-enhanced-Proc* is implemented in CMODELS with SAT solver SIMO or ZCHAFF invoked in place of SAT-A in the procedure. In case of *DP-generate-test-enhanced-Proc* implementation in Step 6 of the algorithm when control is given back to the SAT solver SIMO or ZCHAFF their behaviour is slightly different. SIMO continues its work with the same search tree it obtained in previous computations, while ZCHAFF starts building a new search tree.

The next experiment that we show is the case of nontight disjunctive 2QBF benchmark. The problem is Σ_2^P -hard. The encoding and the instances of the problem were obtained at the web-site of Logic and Artificial Intelligence Laboratory of the University of Kentucky⁴. Figure 3 presents the results. The experiments were run on Pentium

² <http://www.princeton.edu/~chaff/>

³ <http://www.star.dist.unige.it/~sim/simo/>

⁴ <http://www.cs.uky.edu/ai/benchmark-suite/>

instance	satisfiability	dlv.5.02.23	cmodels mchaff	cmodels zchaff	cmodels simo	gnt2
qbf7	SAT	15.67	0.01 (23)	0.01 (16)	0.14 (5)	-
qbf8	SAT	92.45	0.01 (23)	0.01 (5)	0.09 (4)	-
qbf9	SAT	7.50	0.01 (33)	0.01 (12)	0.09 (5)	25.77
qbf1	UNSAT	19.81	0.21(10)	0.01 (16)	0.01 (37)	0.001
qbf2	UNSAT	5.43	-	823.98 (19928)	239.68 (26523)	1466.30
qbf3	UNSAT	5.27	-	1779.28 (28481)	193.69 (21260)	-
qbf4	UNSAT	6.83	memory	10.55 (137)	33.64 (663)	-

Fig. 3. CMODELS using MCHAFF, ZCHAFF, SIMO vs. DLV, and GNT on 2QBF benchmark

4, CPU 3.00GHz. The columns 3 through 7 present the running times of the systems in seconds with 30 minutes cutoff time. Number in parenthesis specifies how often CMODELS invoked the minimality test procedure during its run. In case of satisfiable instances of the problem we can clearly see the payoff in using system CMODELS in place of other disjunctive ASP solvers. It is faster than DLV by several orders of magnitude. The picture changes when unsatisfiable instances of the problem come into play. Implementation of *DP-assat-Proc* reaches time limit twice and in case of one instance reaches the memory limit. Implementation of *DP-generate-test-enhanced-Proc* shows better results but as a rule is slower than DLV running time by two orders of magnitude. If we pay attention to the number of minimality test procedure invocations, the slow performance is then not surprising. The number of models of completion is large in case of *qbf2*, *qbf3* instances which are unsatisfied and hence all models found by the SAT solver must be verified and denied by the minimality test procedure.

The last experiment that we demonstrate is the case of nontight disjunctive Strategic Company benchmark. The problem is Σ_2^P -hard. We used the encoding and the instances of the problem provided by the benchmark system for answer set programming – Asparagus⁵. Figure 4 presents the running times of the answer set programming system obtained from Asparagus that uses machine AMD Athlon 1.4GHz PC with 512MB RAM and cutoff time 15 minutes while running the experiments. All given instances are satisfiable. In case of strategic company benchmark there is no clear winner in the performance, but GNT and DLV are in general faster.

7 Proofs

Lemma 1. *Let Π be a DP, M be the model of $Comp(\Pi)$. If $a \in M$ then there must be a supporting rule $A \leftarrow Body$ in Π for a under M .*

Proof. From the completion construction there must be a clause

$$a \supset \bigvee_{A \leftarrow Body \in \Pi, a \in A} (Body \wedge \bigwedge_{p \in A \setminus \{a\}} \neg p)$$

⁵ <http://asparagus.cs.uni-potsdam.de/>

inst- ance	dlv.4 5.23	gnt2	cmodes zchaff	cmod-s mchaff	cmod-s simo	inst- ance	dlv.4 5.23	gnt2	cmodes zchaff	cmod-s mchaff	cmod-s simo
160.1	0.64	1.08	0.33	0.40	0.34	125.45	9.03	41.02	-	-	-
160.3	0.87	1.23	0.34	0.40	0.34	105.38	15.55	79.99	315.41	404.72	580.23
75.37	0.51	6.78	1.20	2.49	1.49	155.0	26.15	16.56	-	-	-
150.2	6.66	41.25	1.52	2.10	5.04	135.11	49.01	8.00	191.89	62.25	577.12
150.26	2.24	5.64	5.99	27.04	14.27	155.3	144.00	188.14	43.11	755.12	215.46

Fig. 4. CMODELS using ZCHAFF, MCHAFF, SIMO vs. DLV, GNT on Strategic Company benchmark.

in $Comp(\Pi)$. This clause is satisfied by M and therefore there exists at least one rule $A \leftarrow Body$ such that $a \in A$, $M \models Body$, and $A \cap M = \{a\}$. Such rule is a supporting rule for a under M .

Lemma 2. *Let Π be a DP, set of atoms $M \models \Pi^M$, set of atoms $M' \models \Pi^M$, such that $M' \subset M$. Any supporting rule of atom $a \in M \setminus M'$ under M contains some atom b in its body such that $b \in M \setminus M'$.*

Proof. Suppose there exists supporting rule $A \leftarrow B, F$ of the form (1) of atom $a \in M \setminus M'$ under M such that B does not contain any atom $b \in M \setminus M'$, in other words $B \subseteq M'$. $M \models B, F$ by definition of a supporting rule, hence rule $A \leftarrow B, F^M \in \Pi^M$, where F^M is a conjunction of \top 's. We are given that $M' \models \Pi^M$, hence $M' \models A \leftarrow B, F^M$. Since $B \subseteq M'$, $M' \models B, F^M$ hence $M' \models A$ and $a \in M'$. Here we derive to contradiction.

Lemma 3. *Let Π be a DP, M be a model of $Comp(\Pi)$, set of atoms $M' \models \Pi^M$, such that $M' \subset M$. Let $G_{M \setminus M'}$ be a subgraph of the positive dependency graph of Π induced by $M \setminus M'$. Then for any $a \in M \setminus M'$, there must be maximal loop $L \subseteq M \setminus M'$ and for some $b \in L$, there is a directed path from a to b in $G_{M \setminus M'}$.*

Proof. From Lemma 1 each atom $a \in M \setminus M'$ has a supporting rule under M . From Lemma 2 it follows that each supporting rule of $a \in M \setminus M'$ has an atom $b \in M \setminus M'$ in its body. From the construction of $G_{M \setminus M'}$ for each $a \in M \setminus M'$ there must be an arc (a, b) where $b \in M \setminus M'$. a is any node and $G_{M \setminus M'}$ has finite number of nodes, so this must lead to a cycle and thus to a strongly connected component L reachable from a . By definition L is a maximal loop under $M \setminus M'$, and hence $L \subseteq M \setminus M'$.

Proposition 1. *Let Π be a DP, M a model of $Comp(\Pi)$, and M' a model of Π^M , such that $M' \subset M$. There must be a terminating loop of Π under $M \setminus M'$. Furthermore, M does not satisfy the loop formula of any of the terminating loops of Π under M .*

Proof. Let $G_{M \setminus M'}$ be a subgraph of the positive dependency graph of Π induced by $M \setminus M'$. From Lemma 3 it follows that there exists maximal loop in $G_{M \setminus M'}$. Clearly, if there exists a maximal loop in a graph then there exists a terminating loop. Assume L is a terminating loop of Π under $M \setminus M'$. Its loop formula is of the form (5). Suppose that the model M satisfies the loop formula of L . There exists at least one rule of the

form (1) — $A \leftarrow B, F$, where $A \cap L \neq \emptyset$, and $B \cap L = \emptyset$, such that

$$M \models B, F \wedge \bigwedge_{p \in A \setminus L} \neg p$$

Assume $A \leftarrow B, F$ is such a rule. There are two cases:

- 1 $(M \setminus M') \cap B \neq \emptyset$. Let b be an atom such that $b \in (M \setminus M') \cap B$. Atom $b \notin L$ due to the condition $B \cap L = \emptyset$ on $A \leftarrow B, F$.

By Lemma 3 there must be maximal loop L_1 such that there is a directed path from b to some atom $l \in L_1$. L_1 must be different from L , since $b \notin L$. $A \cap L \neq \emptyset$ is the condition on $A \leftarrow B, F$. Let atom a be an atom such that $a \in A \cap L$. From $G_{M \setminus M'}$ construction node corresponding to atom a has an edge to node corresponding to atom b , and hence has a directed path to l . This contradicts to our assumption that L is terminating.

- 2 $(M \setminus M') \cap B = \emptyset$. Hence $B \subseteq M'$. $M \models B, F$ from the assumption, and hence $A \leftarrow B, F^M \in \Pi^M$, where F^M is the conjunction of \top . We are given that $M' \models \Pi^M$ therefore $M' \models A \leftarrow B, F^M$. $M' \models A$ must hold since $B \subseteq M'$. From our assumption $M \models \bigwedge_{p \in A \setminus L} \neg p$, and given $M' \subset M$ we derive that $M' \models \bigwedge_{p \in A \setminus L} \neg p$.

From conclusions that (i) $M' \models A$ and (ii) $M' \models \bigwedge_{p \in A \setminus L} \neg p$ we derive $M' \models a$,

such that $a \in A \cap L$. Hence $L \cap M' \neq \emptyset$. This contradicts to our assumption that $L \subseteq M \setminus M'$.

8 Conclusions

Systems ASSAT and CMODELS are the implementations that demonstrated promising experimental results of the SAT based approach for the case of nondisjunctive programs. In this work we described the theoretical background for the extending the implementation to the case of disjunctive programs as well as provided the implementation itself. Preliminary experimental results support the evidence that the approach may promote the use of disjunctive answer set programming in practice. Our implementation at the same time introduced the new feature among current answer set programming systems as allowing to use disjunctive and choice rules in the same programs.

Acknowledgements

I would like to thank Enrico Giunchiglia, Guenther Goerz, Joohyung Lee, Liu Lengning, Nicola Leone, Vladimir Lifschitz, Marco Maratea, Gerald Pfeifer, Mirek Truszczyński for the comments related to the subject of this paper, and Tommi Syrjänen for implementing `--dlp-choice` option in front-end LPARSE.

References

- [BEMR05] D. R. Brooks, E. Erdem, J. W. Minett, and D. Ringe. Character-based cladistics and answer set programming. In *Proc. PADL'05*, pages 37–51, 2005.

- [Cla78] Keith Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.
- [DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.
- [EL03] Esra Erdem and Vladimir Lifschitz. Tight logic programs. *Theory and Practice of Logic Programming*, 3:499–518, 2003.
- [EW04] E. Erdem and M.D.F. Wong. Rectilinear steiner tree construction using answer set programming. In *Proc. ICLP'04*, pages 386–399, 2004.
- [FL05] Paolo Ferraris and Vladimir Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 5:45–74, 2005.
- [GL91] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.
- [GLM04] Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. Sat-based answer set programming. In *Proc. AAI-04*, pages 61–66, 2004.
- [Jea05] T. Janhunen and et al. *GnT (Generate'n'Test): A Solver for Disjunctive Logic Programs*. 2005. Available under <http://www.tcs.hut.fi/Software/gnt/>.
- [JNSY00] T. Janhunen, I. Niemela, P. Simons, and J.H. You. Unfolding partiality and disjunctions in stable model semantics. In *Proc. KR*, 2000.
- [KLP03] C. Koch, N. Leone, and G. Pfeifer. Enhancing disjunctive logic programming systems by sat checkers. *Artificial Intelligence*, 151:177–212, 2003.
- [Le05] Yu. Lierler and etc. *CMODELS – a tool for computing answer set using SAT solvers*. 2005. Available <http://www.cs.utexas.edu/users/tag/cmodels>.
- [Lie05] Yu. Lierler. Cmodels for tight disjunctive logic programs. In *19th Workshop on (Constraint) Logic Programming W(C)LP*, number 2005-01 in Ulmer Informatik-Berichte, 2005. <http://www.informatik.uni-ulm.de/epin/pw/11541>.
- [LL03] Joohyung Lee and Vladimir Lifschitz. Loop formulas for disjunctive logic programs. In *Proc. ICLP-03*, pages 451–465, 2003.
- [LM04] Yuliya Lierler and Marco Maratea. Cmodels-2: Sat-based answer set solver enhanced to non-tight programs. In *Proc. LPNMR-04*, pages 346–350, 2004.
- [LPE⁺05] N. Leone, G. Pfeifer, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The dlv system for knowledge representation and reasoning. *ACM TOCL*, 2005. To appear. System available under <http://www.dbai.tuwien.ac.at/proj/dlv/>.
- [LZ02] Fangzhen Lin and Yuting Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. In *Proc. AAI-02*, pages 112–117, 2002. System ASSAT available at <http://assat.cs.ust.hk/>.
- [NS00] Ilkka Niemelä and Patrik Simons. Extending the Smodels system with cardinality and weight constraints. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 491–521. Kluwer, 2000.
- [SS05] P. Simons and T. Syrjaenen. *S MODELS and LPARSE – a solver and a grounder for normal logic programs*. 2005. Available at World Wide Web <http://saturn.hut.fi/pub/smodels/>.
- [WS04] J. Ward and J. Schlipf. Answer set programming with clause learning. In *Logic Programming and Nonmonotonic Reasoning, 7th International Conference*, 2004.