2005

CMODELS – SAT-based Disjunctive Answer Set Solver

Yuliya Lierler
University of Nebraska at Omaha, ylierler@unomaha.edu

CMODELS – SAT-based Disjunctive Answer Set Solver

Yuliya Lierler

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

Introduction

Disjunctive logic programming under the stable model semantics [GL91] is a new methodology called *answer set programming* (ASP) for solving combinatorial search problems. This programming method uses answer set solvers, such as DLV [Lea05], GNT [Jea05], SMODELS [SS05], ASSAT [LZ02], CMODELS [Lie05a]. Systems DLV and GNT are more general as they work with the class of disjunctive logic programs, while other systems cover only normal programs. DLV is uniquely designed to find the answer sets for disjunctive logic programs. On the other hand, GNT first generates possible stable model candidates and then tests the candidate on the minimality using system SMODELS as an inference engine for both tasks. Systems CMODELS and ASSAT use SAT solvers as search engines. They are based on the relationship between the completion semantics [Cla78], loop formulas [LZ02] and answer set semantics for logic programs. Here we present the implementation of a SAT-based algorithm for finding answer sets for disjunctive logic programs within CMODELS. The work is based on the definition of completion for disjunctive programs [LL03] and the generalisation of loop formulas [LZ02] to the case of disjunctive programs [LL03]. We propose the 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 CMODELS and demonstrate the experimental results.

1 Syntax of CMODELS

A *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 $\bot$, B is a conjunction of atoms, and F is a formula of the following form

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

where A_i are atoms. We call such rules *disjunctive*. If a head of a rule does not contain disjunction, we call such a rule *normal*. If the formula F of the rule (1) contains an expression of the form $not\ not\ A_i$ then the rule is *nested*, otherwise the rule is *non-nested*. If all rules of a DP are normal we call the program normal.

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

$$\{A_0, ...A_k\} \leftarrow A_{k+1}, \ldots, A_l, not\ A_{l+1}, \ldots, not\ A_m \tag{2}$$

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

$$A_0 \leftarrow L[A_1 = w_1, \ldots, A_m = w_m, not\ A_{m+1} = w_{m+1}, \ldots, not\ A_n = w_n] \tag{3}$$

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

The concept of an answer set for programs containing rules (2) and (3) was introduced in [NS00]. The original rules given to the front end LPARSE *--dlp-choice* allow lower and upper bounds for choice rules and upper bounds for weight rules. They also allow use of literals (negated atoms) in place of atoms. LPARSE *--dlp-choice* translates all the rules to the forms specified above. In CMODELS, choice rules are translated into normal nested rules, and weight constraints are translated with the help of auxiliary variables into normal non-nested rules.[FL05]

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

2 Details on the Modified Algorithms and the Implementation

The implementation is based on definitions of completion, tightness and loop formula for DP introduced in [LL03]. We also refer the reader to [LL03] for formal definitions of a set of atoms satisfying a program, answer set, reduct, and positive dependency graph of DP. The implementation exploits the relationship between completion semantics, loop formulas and answer set semantics for DP. For class of programs called tight models of completion and answer sets are the same. For nontight programs the difference in semantics is due to the cycles (loops) in the program. Loop formulas serve a role of an extension to completion so that the semantics coincide again. Number of loop formulas is exponential and therefore precomputing all loop formulas at once is not feasible, and iterative approach is explored. The correctness of algorithms encoded in CMODELS follows from two theorems.

Theorem for Tight Programs. [LL03] *For any tight DP Π and any set X of atoms, X is an answer set for Π iff X satisfies program's completion $comp(\Pi)$.*

Theorem 1. *Let Π be a DP, M be a model of its completion $comp(\Pi)$, set of atoms $M' \models \Pi^M$, such that $M' \subset M$. There must be a loop of Π under $M \setminus M'$, s.t. M does not satisfy its loop formula.*

Deciding whether a model of the completion is an answer set of disjunctive program is co-NP-complete. Within this implementation of CMODELS we verify that a model of the completion is indeed an answer set by using the minimality requirement of an answer set. We invoke a SAT solver on formula $\Pi^M \cup M^- \cup \neg M$, where (i) Π^M denotes the reduct of Π under M, s.t. its rules are represented as propositional formulas with the comma understood as conjunction, and $A \leftarrow B$ as the material implication $B \supset A$;

(ii) M^- denotes the conjunction of negation of the atoms in Π that do not belong to M; and (iii) $\neg M$ denotes the negation of the conjunction of atoms in M. If this formula is unsatisfied then M is indeed an answer set of Π otherwise some model $M' \subset M$ is returned. Note that $M' \models \Pi$. We call this procedure *minimality test*. It is similar to the procedure introduced in [JNSY00]. [KLP03] introduced a more sophisticated way of verifying whether a model is an answer set using SAT solvers by exploiting some modularity property of the program, that permits splitting verification step on the whole program into verification on its parts. It is a direction of future work to research the applicability of the approach to the case of nested programs.

CMODELS' algorithm is enhanced to verify the tightness of DP at first. In case when a program is tight it performs a completion procedure on the program and uses a SAT solver for enumerating its answer sets, avoiding invocation of minimality test procedure. 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 nontight programs we adapt ASSAT algorithm [LZ02] to the case of disjunctive programs based on Theorem 2. The modified algorithm follows — *DP-assat-Proc*:

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

The implementation also adapts another SAT-based answer set programming generate and test algorithm from [GLM04] to the case of nontight disjunctive programs. State-of-the-art SAT solvers are enhanced by the ability of performing backjumping and learning within standard SAT Davis-Logemann-Loveland (DLL) procedure. Backjumping and learning techniques are due to providing DLL procedure with a certain clause. We retrieve the necessary clause from some loop formula of a program that allows us to enhance SAT solver inner computation. The enhanced generate and test algorithm for DP — *DP-generate-test-enhanced-Proc*:

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

instance	sat	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. 1. CMODELS using MCHAFF, ZCHAFF, SIMO vs. DLV, and GNT on 2QBF benchmark

3 Experimental Analyses

Details on the performance of system CMODELS in case of tight disjunctive programs can be found in [Lie05b]. For experimental analysis of CMODELS' performance on non-tight programs we shall specify the algorithmic differences of SAT solvers' invocations. Algorithm *DP-assat-Proc* is implemented in CMODELS using SAT solver MCHAFF[1] in Step 2. Algorithm *DP-generate-test-enhanced-Proc* is implemented in CMODELS with SAT solver SIMO[2] or ZCHAFF[1] invoked in place of *SAT-A* in the procedure. In case of *DP-generate-test-enhanced-Proc* implementation of Step 6 when control is given back to the SAT solver, SIMO and ZCHAFF behave differently. SIMO continues its work with the same search tree it obtained in previous computations, while ZCHAFF starts building a new search tree. In all cases ZCHAFF is used for minimality test procedure.

The first experiment that we demonstrate is 2QBF benchmark. The problem is Σ_2^p-hard. The encoding and the instances of the problem where obtained at the web-site of the University of Kentucky [3]. Figure 1 presents the results. The experiments were run on Pentium 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 parentheses specifies how often CMODELS invoked the minimality test procedure during its run. In case of satisfiable instances of the problem we can see the payoff in using CMODELS in place of other disjunctive ASP solvers. 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 not surprising. The number of models of the completion is large in case of unsatisfiable instances *qbf2, qbf3* instances and hence all found models must be verified and denied by the minimality test procedure.

The second experiment that we present is the 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 – Asparagus [4]. Figure 2 presents

[1] `http://www.princeton.edu/~chaff/`
[2] `http://www.star.dist.unige.it/~sim/simo/`
[3] `http://www.cs.uky.edu/ai/benchmark-suite/`
[4] `http://asparagus.cs.uni-potsdam.de/`

running times of systems obtained from Asparagus, machine AMD Athlon 1.4GHz PC with 512MB RAM and cutoff time 15 minutes. 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.

inst-ance	dlv.4 5.23	gnt2	cmodels zchaff	cmod-s mchaff	cmod-s simo	inst-ance	dlv.4 5.23	gnt2	cmodels 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. 2. CMODELS using ZCHAFF, MCHAFF, SIMO vs. DLV, GNT on Strategic Company.

References

[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.

[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. AAAI-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.

[Lea05] N. Leone and et al. *A disjunctive datalog system DLV (2005-02-23)*. 2005. Available under `http://www.dbai.tuwien.ac.at/proj/dlv/`.

[Lie05a] Y. Lierler. *CMODELS – a tool for computing answer set using SAT solvers.* 2005. Available under `http://www.cs.utexas.edu/users/tag/cmodels`.

[Lie05b] Yu. Lierler. Cmodels for tight disjunctive logic programs. In *19th Workshop on (Constraint) Logic Programming W(C)LP*, 2005.

[LL03] Joohyung Lee and Vladimir Lifschitz. Loop formulas for disjunctive logic programs. In *Proc. ICLP-03*, pages 451–465, 2003.

[LZ02] Fangzhen Lin and Yuting Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. In *Proc. AAAI-02*, 2002.

[NS00] Ilkka Niemelä and Patrik Simons. Extending the Smodels system with cardinality and weight constraints. In *Logic-Based Artificial Intelligence*, pages 491–521. 2000.

[SS05] P. Simons and T. Syrjaenen. *SMODELS and LPARSE – a solver and a grounder for normal logic programs.* 2005. `http://saturn.hut.fi/pub/smodels/`.