Efficient SAT-based Answer Set Solver [DRAFT]

by

Zhijun Lin, B.E.

A Dissertation

In

COMPUTER SCIENCE

Submitted to the Graduate Faculty

of Texas Tech University in

Partial Fulfillment of

the Requirements for

the Degree of

DOCTOR OF PHILOSOPHY

December, 2007

CONTENTS

Abstract

Recent research shows that SAT (propositional satisfiability) techniques can be employed to build efficient systems to compute answer sets for logic programs. ASSAT and CMODELS are two well-known such systems that work on normal logic programs. They find an answer set from the full models for the completion of the input program, which is (iteratively) augmented with loop formulas. Making use of the fact that, for non-tight programs, during the model generation, a partial assignment may be extensible to a full model but may not grow into any answer set, we propose to add answer set extensibility checking on partial assignments. Furthermore, given a partial assignment, we identify a class of loop formulas that are "active" on the assignment. These "active" formulas can be used to prune the search space. We also provide an efficient method to generate these formulas. These ideas can be implemented with a moderate modification on SAT solvers. We have developed a new answer set solver SAG on top of the SAT solver MCHAFF. Empirical studies on well-known benchmarks show that in most cases it is faster than the state-of-the-art answer set solvers, often by an order of magnitude. For disjunctive logic programs, the existing SAT-based solvers translate them into propositional formulas based on a complex completion definition, and then make use of loop formulas and SAT solvers to find answer set. In this paper we present a new approach that allows the translation of a program into a formula that is weaker but less complex than the completion. It performs answer set checking on partial assignments. In case a partial assignment is inextensible to an answer set, we use support formulas, which is a generalization of loop formula, to prevent the repetition of the same mistake. Empirical studies on disjunctive logic programs confirm the performance advantage of the new approach.

LIST OF TABLES

LIST OF ALGORITHMS

CHAPTER 1

Introduction

1.1 ASP and SAT

Logic programming with answer sets semantics [8], also called answer set programming (ASP), has emerged as one of the leading language for knowledge representation. It has found practical applications in the areas like planning and diagnosis, model checking and graph algorithms. Answer set programming and propositional satisfiability (SAT) are closely related. It is well-known that an answer set of a logic program is also a model of its completion [3]. The converse holds for tight programs [5]. For non-tight programs, Lin and Zhao [18] show that by adding loop formulas to the completion, one can obtain a one-to-one correspondence between the answer sets of a logic program and the models of its extended completion. Lee and Lifschitz [13] generalize the concept of completion and loop formula for logic programs with nested expressions and disjunctive rules. As a result, two SAT-based answer set solvers were implemented: ASSAT by Lin and Zhao [18] and CMODELS by Lierler and Maratea [17]. ASSAT supports only non-disjunctive (normal)logic programs, CMODELS supports both disjunctive/non-disjunctive programs, as well as programs with cardinality constraint rules.

Both ASSAT and CMODELS look for answer sets of a logic program from the full models of its completion. These models are generated by a SAT solver. Observing that, for non-tight programs, during the model generation, a partial assignment may be extensible to a full model but may not grow into any answer set, we propose a new answer set solving procedure that initiates answer set extensibility checking before a full model is generated. We find that some loop formulas are responsible for the checking results, and they can be further used to prune the search space.

This new approach has been proved very effective by our empirical studies.

We also generalize our idea to apply to answer set solving for logic programs with cardinality constraint rules and disjunctive logic programs. For the treatment of cardinality constraint rules, we use a novice "guess and check" approach instead of the rule translation approach used by the existing SAT-based answer set solver. The advantage of the new approach is that it prevents the introduction of too many extra rules and variables. The disjunctive logic programs have higher expressive power and higher computational complexity. To compute answer set for a disjunctive logic program, the existing solver, CMODELS, uses the similar approach it uses in answer set solving for normal logic program, except a more complex form of completion and loop formulas are used. We observe that the complex completion formula may lead to excessive number of clauses and variables, which has a negative impact on the solver's performance. Our approach uses a less complex form of completion as initial translation of the input program, and uses a new mechanism called support formula to replace loop formula. Combined with allowing answer set extensibility checking for partial assignments, our implementation shows a solid performance advantage over the existing system.

1.2 Summary of contributions

The following are the main contributions of this dissertation.

1. Improve the existing SAT-based answer set solving algorithms by introducing partial answer set checking.

2. Generalize the concept and theories of loop formula to apply to partial assignments, making it possible to take advantage the conflict-driven learning mechanism provided by the modern SAT solvers.

3. Develop a "guess and check" approach to handle cardinality constraint rules, which will not blow-up the number of clauses and variables in the translated boolean formula for SAT solver.

4. Introduce support formula to replace loop formula in answer set solving for disjunctive logic programs, allowing the use of less complex completion forms, thus avoid the slow down caused by the excessive number of clauses and variables.

5. Implement two new SAT-based answer set solvers, SAG for non-disjunctive logic programs, and DSAG for disjunctive logic programs. Both solver are competitive in terms of performance against the state of the art answer set solvers.

1.3 Related works

Other answer set solvers take advantage of morden SAT techniques include SMODELSCC [26] and CLASP [7]. Neither solvers are built upon a SAT solver, instead, SMODELSCC incorporates conflict driven learning techniques of SAT into existing SMODELS framework, while CLASP build a brand new solver from scratch.

1.4 Organization

The rest of this dissertation is organized as follows. First we review background knowledge in answer set programming and boolean satisfiability. Then we present our answer set solving procedure for normal programs using a SAT solver, proof of its correctness is also provided. Next we present the solution for disjunctive programs, especially for the hard instances that are non-head-cycle-free. The solution for programs with choice rules and cardinality constraint rules follows. Finally we report the experimental results before the conclusion.

CHAPTER 2

Background

2.1 Answer Set Programming

In the section, we review the background of logic programming with answer set semantics (also called *A-Prolog*).

2.1.1 Basic syntax and semantics

A *disjunctive rule* is a logic formula of the form

$$a_1 \vee \cdots \vee a_k, \leftarrow a_{k+1}, \ldots, a_m, \mathrm{not}\ a_{m+1}, \ldots, \mathrm{not}\ a_n. \tag{2.1}$$

where a_i's are atoms. For a disjunctive rule r, we denote the set $\{a_1, \ldots, a_k\}$ by $head(r)$, the set $\{a_{k+1}, \ldots, a_m\}$ by $pos(r)$, and $\{a_{m+1}, \ldots, a_n\}$ by $neg(r)$. In this paper, we sometimes use the following form to represent the above rule:

$$head(r) \leftarrow pos(r), \mathrm{not}\ neg(r).$$

A logic program is a finite set of disjunctive rules. If a logic program constrains only rules that have no more than one head atom, it is called a *normal logic program*. A program is positive if for every rule r in the program, $neg(r) = \emptyset$.

Given a logic program P, we use $U(P)$ to denote the set of all atoms appear in P. Given a set of atoms M, if $M \subseteq U(P)$, and for each rule $r \in P$, $pos(r) \not\subseteq M$ or $neg(r) \cap M \neq \emptyset$ or $head(r) \cap M \neq \emptyset$, we say M satisfies r, denoted by $M \models r$.

Example 2.1.1. *Given rule* r :

$$a \vee b \leftarrow c, d, \mathrm{not}\ e, \mathrm{not}\ f$$

$\{a, c, d\} \models r$ *holds, but* $\{b, c, e\} \models r$ *does not.*

A set of atoms M is called a model of program P if M satisfies every rule in P. If P is a positive program and M is the minimum (under set inclusion) model of it, then M is said to be an *answer set* of P.

Example 2.1.2. *Given the program*

$$a \vee b.$$

$$c \leftarrow a.$$

$$d \leftarrow b.$$

$M_1 = \{a, c\}$ *and* $M_2 = \{a, b, c\}$ *are both models, but* $M_2 \supset M_1$ *so* M_2 *is not an answer set.* M_1 *is an answer set because none of* $\emptyset$, $\{a\}$ *and* $\{c\}$ *is a model.*

Given an arbitrary program P and a set of atoms $M \subseteq U(P)$, the reduct of P, denoted by P^M, is the positive program

$$\{head(r) \leftarrow pos(r) \mid r \in P, neg(r) \cap M = \emptyset\}.$$

A set of atoms M is an answer set of program P if and only if M is an answer set of P^M.

Example 2.1.3. *The program* P :

$$a \leftarrow not\ b.$$

$$b \leftarrow not\ a.$$

$$c \leftarrow a, not\ d.$$

$$d \leftarrow not\ c.$$

has an answer set $\{a, c\}$*, which is also an answer set of* $P^A =$

$$a.$$

$$c \leftarrow a.$$

2.1.2 Variables and Functions

The logic programs we described in the previous section use atoms as basic components. This kind of programs are also called *grounded programs*. Writing grounded programs is both tedious and error-prone. The answer set programming language actually support the use of variables and functions. To compute answer sets, a parser is usually used to covert programs with variables and functions to the corresponding grounded programs. In the rest of this paper,est of this paper, we limit our consideration to fully grounded programs.

2.1.3 Classical negation

In [9], *classical negation* operator $\neg$ is introduced to distinguish against *negation as failure* operator *not*. Intuitively, $\neg a$ means a is $false$ while $not\ a$ means there is no reason to believe a is $true$. Syntactically, the extended programs allows $\neg$ operator appear before any atom in a rule. An atom or an atom with $\neg$ operator is referred to as a $literal$. Semantically, we can treat an literal $\neg a$ as a new atom that should never belong to the same answer set as atom a. It is shown in [9] that any program containing classical negation operation can be easily converted to a equivalent program containing only negation as failure operator. In fact, most of the existing grounding programs automatically perform this conversion.

2.1.4 Weight Constraint Rules

Simons, Niemela and Soininen [23] introduced *weigtht constraint rules* as an extension to the standard normal answer set semantics. The purpose is to increase the expression of logic programs to compactly represent constraints like cardinality and weight constraints. The building block of a constraint rule called *weight constraint*, which is of the form

$$l \leq \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, \text{not } b_1 = w_{b_1}, \ldots, \text{not } b_m = w_{b_m}\} \leq u \tag{2.2}$$

where a_i,b_j is an atom. We refer to an atom a or the expression $\text{not } a$ as a *literal*. Each literal in a constraint has an associated weight, e.g., literal $\text{not } b_1$ has weight w_{b_1}. l and u represent the *lower* and *upper bounds* of the constraint, respectively. Either of the bounds can be omitted, which means the default lower bound $-\infty$ or upper bound ∞ is used. Given a set of atom M,

A *weight constraint rule* is an expression of the form

$$C_0 \leftarrow C_1, \ldots, C_n \tag{2.3}$$

Where each C_i is a weight constraint.

In case all weights of a constraint are 1, we can use a short hand expression

$$l\{a_1, \ldots, a_n, \text{not } b_1, \ldots, \text{not } b_m\}u$$

which is also called a *cardinality constraint*.

A set of atoms M satisfies a weight constraint C of the form 2.2, iff $l \leq$

$W(M, C) \leq u$, where

$$w(M, C) = \sum_{a_i \in M} w_{a_i} + \sum_{b_i \notin M} w_{b_i}.$$

A weight constraint rule r of the from 2.3 is satisfied by M ($M \models r$) iff M satisfies C_0 when ever M satisfies every constraint in the rule body.

In [23] it is shown that an arbitrary constraint rule can be rewritten as a combination of *basic constraint rules*, which include two types:

- *weight rule*, which is of the form

$$h \leftarrow l \leq \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, \text{not } b_1 = w_{b_1}, \ldots, \text{not } b_m = w_{b_m}\}$$

  where all weights are non-negative.

- *choice rule*, which is of the form

$$\{h_1, \ldots, h_k\} \leftarrow a_1, \ldots, a_n, \text{not } b_1, \ldots, \text{not } b_m.$$

The reduct of a weight rule r w.r.t. a set of atoms M, denoted by r^M, is the rule

$$l' \leq \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}\}$$

where the lower bound

$$l' = l - \sum_{b_i \notin M} w_{b_i}.$$

The reduct of a choice rule r w.r.t. M is a set of rules

$$\{h_i \leftarrow a_1, \ldots, a_n \mid h_i \in M\}.$$

The reduct of a program P is the set of reduct of each rules in P. A set of atoms M is an answer set of P if $S \models P$ and S is a minimum model of P^S.

Example 2.1.4. *Consider program*

$$\{a, b, c\}.$$

$$d \leftarrow 2 \leq \{a, not\ b, not\ c\}.$$

Set $A = \{a, d\}$ *is an answer set, since* A *is a minimum model of* $P^A =$

$$a \leftarrow$$

$$d \leftarrow 0 \leq \{a\}$$

2.2 Boolean satisfiability

2.2.1 Basic concepts

The *Boolean Satisfiability (SAT)* problem is to find a variable assignment that make a boolean formula evaluate to $True$, or to determine no such assignment exists. The boolean formula are usually specified in conjunctive normal form (CNF). A CNF formula is a conjunction of *clauses*, where each clause is a disjunction of literals.

Example 2.2.1. *The following formula*

$$(a, b)(\neg a, c)(\neg b, d)$$

is satisfied by the assignment $(a(True), b(False), c(True), d(False))$.

In the rest of this paper, we will use set of literals to represent variable assignment. For example, the assignment in the previous example will be written as $\{a, \neg b, c, \neg d\}$.

All boolean formula can be described in CNF format, which has an advantage: for the formula to be satisfied (sat), each $clause$ must be sat. But the CNF representation lacks the benefit compactness and intuitiveness of general boolean formula. So in the rest of discussion, we will represent boolean formulas in general form using operators besides conjunction and disjunction. Just remember they will be convert to CNF form.

2.2.2 Logic program and boolean formula

By mapping each disjunctive rule of a logic program that is of the form

$$a_1 \vee \cdots \vee a_k, \leftarrow a_{k+1}, \ldots, a_m, \mathrm{not}\ a_{m+1}, \ldots, \mathrm{not}\ a_n$$

into a boolean formula of the form

$$(a_1 \vee \cdots \vee a_k) \subset (a_{k+1} \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n),$$

it can be verified the model of the original program corresponds to the sat assignment for the derived formula. In the later sections we will investigate the relationship between answer set finding and SAT problems. But first let us take a brief review of basic SAT search procedure.

2.2.3 SAT search

Most of the successful complete SAT solvers are based on Davis Putnam Logemann Loveland (DPLL) algorithm [4]. Recently, it has been shown that learning techniques play a significant role in improving the performance of SAT solvers (e.g., MCHAFF[19], Rel_sat[1] and GRAPS [22]). Algorithm 1 illustrates a typical DPLL algorithm with learning that is adapted from [27].

Algorithm 1: DPLL algorithm with learning

```
DPLL-learning(F)
1  // F is a boolean formula with conjunctive normal form
2  lclauses ← ∅, blevel ← 0
3  if deduce (F, ∅)=conflict then return no answer set
4  while true do
5      let B be the current assignments
6      if there exists a free variable then
7          (x, a) ← decide (F, B)
8          blevel ← blevel + 1
9          B ← B ∪ {x = a}
10         while deduce (F∪ lclauses, B)=conflict do
11             cc ← gen_conflict_clauses (F∪ lclauses, B)
12             blevel ← find_blevel (cc)
13             lclauses ← lclauses + cc
14             if blevel =0 then
15                 return unsatisfiable
16             else
17                 B ← backtrack (blevel)
18     else
19         return the model B
```

Given an input propositional formula F, the algorithm DPLL-learning uses a backtracking search to find a truth assignment to the variables of F such that F is evaluated to be true. Such an assignment is called a model of F. It returns unsatisfiable if no such an assignment exists (line 15), and an assignment otherwise (line 19). Given F and the current (partial) assignment B, the function decide(F, B) returns a free variable x and a truth assignment a for x based on some heuristics (line 7). The decision level blevel is then increased by 1 (line 8). Here, we do not cover the details of blevel that is part of the backtracking mechanism. After a is assigned to x (line 9), the function deduce is called to prune the search space using F, lclauses (explained later) and the current assignment (line 10). For those variables with only one truth value left after pruning, deduce assigns those variables with the

corresponding values. The function deduce is based on *unit propagation*. It returns conflict if a variable needs to be both $true$ and $false$. In this case, the function gen_conflict_clauses is invoked to analyze the reasons for the conflict and generate clauses cc to remove the conflict (line 11). The function find_blevel uses cc to find a proper decision level to backtrack to (line 12) so that the current conflict can be resolved. The function backtrack(blevel) restores the search to the state at the decision level of blevel and returns the current assignment at that level (line 17). The clauses cc are called conflict clauses. They are added to the buffer of lclauses (line 13) to prevent the same conflict from happening in the future search. This is regarded as a learning, and the conflict clauses are sometimes called learning clauses. Note the conflict clauses in lclauses are redundant to F. In practice, SAT solvers impose a limit on the size of lclauses and, if necessary, discard conflict clauses deemed less useful. Due to two reasons, we ignore here the details of the functions of decide, deduce, gen_conflict_clauses, find_blevel, backtrack, and the management of lclauses. First, there exist excellent references on the algorithm of DPLL with learning (e.g., [27]). More importantly, we intentionally take those functions as black boxes and simply reuse them in designing the ASP solver in this paper, a big advantage offered by using existing SAT solvers.

CHAPTER 3

SAT-based answer set solving for normal logic programs

As we have described in section 2.2.2, we can directly transform a normal logic program into a boolean formula, whose sat-assignments corresponds to the model of the original logic program. In this section, we study ways to use SAT techniques to generate answer sets.

3.1 Clark completion and loop formula

Given a normal logic rule of the form

$$a_0 \leftarrow a_1, \ldots, a_m, \ldots, \mathit{not}\ a_{m+1}, \ldots, \mathit{not}\ a_n \tag{3.1}$$

we use $BC(r)$ to denote the boolean formula

$$a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n.$$

In the special case when $n = 0$, $BC(r) = \mathit{True}$.

The *Clark completion* of a logic program P, denoted by $\mathit{Comp}(P)$, is the set of the following propositional clauses:

- For each atom $a \in \mathit{Atoms}(P)$, let $R = \{r \mid r \in P,\ \mathit{head}(r) = a\}$.
  - If $R = \emptyset$, $(\neg a)$.
  - Otherwise, $(a \equiv \bigvee_{r \in R} BC(r))$.
- For each rule $r \in P$ such that $\mathit{head}(r) = \emptyset$, $(\neg BC(r))$.

The *dependency graph* of a logic program P, denoted by $DG(P)$, is the directed

graph (V, E) where $V = U(P)$ and

$$E = \{(a, b) \mid a, b \in V, r \in P, \{a\} = head(r), b \in pos(r)\}.$$

A set of atoms $L \subseteq V$ is called a loop of P if there is a path between any two members of L without passing any vertex outside L. We say P is tight if $DG(P)$ has no loops; Otherwise, we say it is non-tight. Given a loop L of P, a rule $r \in P$ is called an *external supporting rule* for L if $head(r) \in L$ and $pos(r) \cap L = \emptyset$. Let R be the set of all external supporting rules for L. The *loop formula* associated with L is the clause

$$\bigwedge_{r \in R} \neg BC(r) \supset \bigwedge_{p \in L} \neg p. \tag{3.2}$$

The following theorem describes the relation between the answer set of a logic program and the model of its completion extended by loop formulas.

Theorem 1. *[18] Let* P *be a logic program,* $Comp(P)$ *its completion, and* LF *the set of loop formulas associated with the loops of* P*. We have that for any set of atoms, it is an answer set of* P *iff it is a model of* $Comp(P) \cup LF$*.*

3.2 The existing approach

From Theorem 1, we can envision a straightforward procedure to compute answer sets for a logic program P: First compute all loop formulas and add them to $Comp(P)$, then use a SAT solver to generate the models for the extended completion. However, this approach is not feasible for general programs since they may have exponential number of loops. For this reason, the existing SAT-based answer set solvers (e.g., ASSAT and CMODELS) use a "generate and test" approach, which can be summarized by the procedure illustrated in Algorithm 2. It first generates a model for the completion of the input program (line 4). If no such model exists

there is no answer set (line 6). Otherwise, if the model is an answer set (line 8), it returns the answer set represented by the model (line 9). If it is not an answer set then the procedure computes some loop formulas violated by the model and add them to the completion (line 11–12). This process is repeated until it finds an answer set or reports no solution. When the answer set test fails, to find a new model, ASSAT has to start the underlying SAT solver from scratch (black-box approach), while CMODELS just initiates a backtrack within the SAT solver (clear-box approach). For ASSAT, the loop formulas added at the end of each iteration (line 12) are critical to the correctness of the algorithm, but for CMODELS they are added as learning clauses solely to speed up the search. Therefore, CMODELS' approach reduces the time cost to find a new model and eliminates the needs of space to keep all loop formulas that can be exponential in number.

Algorithm 2: Existing SAT-based answer set solving procedure

```
ASSAT(P)
 1  // P is a logic program
 2  C ← Comp(P)
 3  while true do
 4      Find a model M for C using a SAT solver
 5      if no such M exists then
 6          return no answer set
 7      else
 8          if isAnswerSet (P, M) then
 9              return the answer set M
10          else
11              compute loop formulas F that are not satisfied by M
12              C ← C ∪ F
```

3.3 The new approach

Given a logic program P, both ASSAT and CMODELS look for answer sets of P from the full models for Comp(P), extended with some loop formulas. We observe

that during the model generation, it is possible to detect that a partial assignment is not extensible to an answer set, long before it grows into a full model.

Example. Consider the program $P_1 = \{a \leftarrow b.\ b \leftarrow a.\ a \leftarrow \ not\ c.\ c \leftarrow \ not\ a.\ c \leftarrow b.\}$. Its completion is $\{(a \equiv (b \vee \neg c)), (b \equiv a), (c \equiv (\neg a \vee b))\}$. The partial assignment $\{b, c\}$ is extensible to a model $\{a, b, c\}$ for $Comp(P_1)$, but cannot be extended to any answer set of P_1. □

Based on the above idea, we develop a new SAT-based answer set solving procedure SATASP. It is listed in Algorithm 3 where the functions decide, deduce, gen_conflict_clauses, find_blevel, backtrack, and the management of lclauses are the same as those in Algorithm 1.

SATASP first computes the completion (in CNF form) of the input program P (line 1). It then selects a variable x and an assignment a for it by using decide (line 8–10). The standard SAT operations are applied to the new assignment (line 12–19). If no conflict is detected by deduce (line 12, 20–32), a new function ASP_deduce will check if the current partial assignment B is extensible to an answer set of P (line 21). ASP_deduce returns conflict, implication, or nil. It returns conflict if the current partial assignment cannot be extended to any answer set of P, implication if some free variables of F are required by the answer set semantics to take particular truth values, or nil for all other cases.

If nil is returned, no new information is obtained and SATASP continues to extend the current partial assignment (line 23).

If conflict or implication is returned, SATASP invokes the function gen_inferred_clauses (line 25) to derive clauses to explain the occurrence of the conflict or implication situation. These clauses are called *inferred clauses*. Making use of the mechanism of lclauses of a SAT solver, they are added to lclauses to facilitate the future conflict learning (line 13) and backjumping (line 14,19; 28,32). They are redundant and

thus can be discarded by the mechanism of lclauses. In the case of conflict, find_blevel uses the inferred clauses ic to determine the decision level to backtrack to (line 28). In either case of implication or conflict, new information, i.e., the inferred clauses ic, is obtained. SATASP uses deduce to propagate the consequences of ic (line 12). This process will be repeated (by the while loop of line 11–32) until no model is found (line 17) or no new information is derived by ASP_deduce (line 22–23).

In the following two sections, we show how to implement ASP_deduce and how to generate the inferred clauses.

3.4 Implement ASP_deduce

Currently we use the Atmost operator [23] of SMODELS to implement the ASP_deduce function.

Given a set of literals M and a rule r, the *generalized reduct* of r, denoted by $r^{(M)}$, is

1. $\emptyset$, if $head(r) \cap M^- \neq \emptyset$ or $pos(r) \cap M^- \neq \emptyset$
   or $neg(r) \cap M^+ \neq \emptyset$;

2. $head(r) \leftarrow pos(r)$, otherwise.

Given a logic program P and a set of literals B, $P^{(B)}$ is defined as $\{r^{(B)} \mid r \in P\}$. $\mathsf{Atmost}(P, B)$ is the minimal model for $P^{(B)}$.

Example. Consider the program P_1 again and the partial assignment $B = \{c\}$. We have $P^{(B)} = \{a \leftarrow b.\ b \leftarrow a.\ c.\}$, $\mathsf{Atmost}(P, B)$=$\{c\}$, $N = \{a, b\}$, $N \cap B^+ = \emptyset$ but $N \not\subseteq B^-$. Hence, ASP_deduce returns implication because a and b should be set to false according to the answer set semantics. □

Algorithm 3: New SAT-based answer set solving procedure

```
SATASP(P)
// P is a logic program
F ← Comp(P)
lclauses ← ∅, blevel ← 0
if deduce (F, ∅)=conflict then return no answer set
while true do
    let B be the current assignments
    if there exists a free variable then
        (x, a) ← decide (F, B)
        blevel ← blevel + 1
        B ← B ∪ {x = a}
        while true do
            if deduce (F∪ lclauses, B)=conflict then
                cc ← gen_conflict_clauses (F∪ lclauses, B)
                blevel ← find_blevel (cc)
                lclauses ← lclauses + cc
                if blevel =0 then
                    return no answer set
                else
                    B ← backtrack (blevel)
            else
                status ← ASP_deduce(P, B)
                if status = nil then
                    break
                else
                    ic ← gen_inferred_clauses (P, B)
                    lclauses ← lclauses + ic
                    if status = conflict then
                        blevel ← find_blevel (ic)
                        if blevel =0 then
                            return no answer set
                        else
                            B ← backtrack (blevel)
    else
        return the answer set represented by B
```

Algorithm 4: Function ASP_deduce

```
ASP_deduce (P, B)
1 // P is a logic program, and B a partial assignment
2 N ← Atoms(P) - Atmost (P, B)
3 if N ∩ B⁺ ≠ ∅ then
    status ← conflict
4 else
5     if N ⊆ B⁻ then
          status ← nil
6     else
          status ← implication
7 return status
```

3.5 Compute inferred clauses

In this section, we will present how to compute the inferred clauses from an input logic program and a partial assignment. This section is concluded by the proof of the correctness of the new procedure SATASP.

Definition 2. A loop formula LF associated with a loop L is *active* on a partial assignment B if B satisfies the antecedent of LF and $L \cap B^- = \emptyset$.

Example. Consider the program P_1 again. It has a loop $L = \{a, b\}$. The loop formula associated with L is LF: $c \supset \neg a \wedge \neg b$. It is clear that LF is active on the partial assignment $B = \{c\}$. □

In the new procedure SATASP, the active loop formulas are used to generate the inferred clauses when ASP_deduce returns conflict or implication.

We present the results on ASP_deduce and active loop formulas before an algorithm for the gen_inferred_clauses function.

The following result is useful to understand and prove Theorem 4.

Proposition 3. *Given a positive program* P *and a non-empty set of atoms* L *such that, for any rule* $r \in P$*, if its head is in* L *then its body contains an atom of* L*, the minimal model for* P *does not contain any atom of* L*.*

Proof. Prove by contradiction. Assume M is a minimal model for P and contains some atom from L. Let $M' = M - L$. For every rule $r \in P$, either the body of r contains some atom of L or not. In the former case, $\mathrm{pos}(r) \not\subseteq M'$ and therefore $M' \models r$. In the latter case $\mathrm{head}(r) \notin L$. $M' \models r$ because r does not have any atom from L, $M' = M - L$, and $M \models r$. Therefore, M' is a model for P, contradicting to the minimality of M. □

Theorem 4. *Given a logic program* P *and a partial assignment* B*, if there is a loop formula active on* B*, then ASP_deduce(*P, B*) returns either conflict or implication.*

Proof. Let LF be a loop formula active on B, and $L = \{L_1, \ldots, L_n\}$ be the associated loop. For any rule $r \in P$ such that $\mathrm{head}(r) \in L$ and $\mathrm{pos}(r) \cap L = \emptyset$, $r^{(B)} = \emptyset$ because B falsifies $BC(r)$, i.e., $\mathrm{pos}(r) \cap B^- \neq \emptyset$ or $\mathrm{neg}(r) \cap B^+ \neq \emptyset$. Therefore, for any rule r in $P^{(B)}$, if the head of r is in L then its body must contain an atom from L. By Proposition 3, the minimal model M of $P^{(B)}$, i.e., $\mathrm{Atmost}(P, B)$, does not contain any atom of L. Let $N = \mathrm{Atoms}(P) - M$. $L \subseteq N$ because $L \subseteq \mathrm{Atoms}(P)$ and $L \cap M = \emptyset$. If B^+ contains an atom of L, $N \cap B^+ \neq \emptyset$, and thus ASP_deduce returns conflict (line 3 of Algorithm 4). Otherwise, $N \not\subseteq B^-$ since $L \cap B^- = \emptyset$ (by the definition of the active loop formula) and $L \subseteq N$; hence, ASP_deduce returns implication (line 6). □

Given a directed graph $G = (V, E)$ and a set of vertices $L \subseteq V$, L is called a *terminating loop* if L is a strongly connected component of G and there is no arc from any vertex in L to any vertex outside L.

This definition of terminating loop is similar to the one in [18]. The following property on graphs is needed in the proof of Theorem 7.

Proposition 5. *Given a directed graph with finite vertices, if every vertex has an outgoing arc, there is a cycle in the graph.*

Proof. Let the graph be G with the set of vertices V. Let $p = (v_1, v_2, \ldots, v_{k-1}, v_k)$, $k \leq |V|$, be one of the longest simple paths of G. Let (v_k, v) be an outgoing arc of v_k. Since p is the longest simple path, $v \in \{v_1, \ldots, v_k\}$. Therefore, path $(v_1, \ldots, v_k, v)$ contains a cycle. □

Proposition 6. *Given a directed graph with finite vertices, if every vertex has an outgoing arc, there is a terminating loop in the graph.*

Proof. Let the graph be G. By proposition 5, G has a cycle and thus a strongly connected component (SCC). Let $Q = \{Q_1, Q_2, \ldots, Q_k\}$ be the set of all SCCs of G, and V_1 the set of vertices not in any SCC. Assume G has no terminating loop, i.e., for any Q_i there is an arc from Q_i to some vertex $v \notin Q_i$. Construct a new graph $G' = (V', E')$ where $V' = V_1 \cup Q$ and E'=$\{(x, y) \mid x, y \in V'$, there is an arc from x to y (or a vertex of y if $y \in Q$) in $G\}$. Clearly every vertex in V' has an outgoing arc. By Proposition 5, G' has a cycle C. C does not contain any vertex v of V_1 because otherwise v will belong to some SCC of G. So C contains at least two vertices Q_i and Q_j from Q, contradicting the fact that Q_i and Q_j are strongly connected components. □

Given a logic program P and a partial assignment B, B is *stable* w.r.t. P if $B^+ \cap B^- = \emptyset$, and for any atom $a \in \mathrm{Atoms}(P)$, $\neg a \in B$ iff either there is no rule headed by a, or for any rule $r \in P$ headed by a, $\mathrm{pos}(r) \cap B^- \neq \emptyset$ or $\mathrm{neg}(r) \cap B^+ \neq \emptyset$.

Theorem 7. *Given a logic program* P *and a stable partial assignment* B*, let* $M = \mathrm{Atoms}(P) - B^- -$ *Atmost*(P, B)*. If ASP_deduce*(P, B) *returns conflict or implication, then there is a terminating loop in the subgraph* Q *of* $DG(P^{(B)})$ *induced by* M*. Furthermore, for any terminating loop in* Q*, the associated loop formula is active on* B*.*

Proof. Since ASP_deduce(P, B) returns conflict or implication (line 3, 6 of Algorithm 4), $M \neq \emptyset$. Let $C = \text{Atmost}(P, B)$.

Claim 1: For all $a \in M$, there exists a rule $r \in P^{(B)}$ such that $head(r) = a$ and $pos(r) \cap M \neq \emptyset$.

Consider any $a \in M$. We know $a \notin B^-$ and $a \notin C$. Since B is stable, $a \notin B^-$ implies that there exists a rule $r \in P$ such that $head(r) = a$ and $pos(r) \cap B^- = \emptyset$ and $neg(r) \cap B^+ = \emptyset$. Clearly $r^{(B)} \neq \emptyset$ and $head(r^{(B)}) = a$. Therefore, $r^{(B)}$ is a rule in $P^{(B)}$ headed by a. Since $a \notin C$, and C is the minimal model for $P^{(B)}$, $pos(r) \not\subseteq C$, which, together with $pos(r) \cap B^- = \emptyset$, implies $pos(r) \cap M \neq \emptyset$.

Claim 2: For every rule $r \in P^{(B)}$, if $head(r) \in M$ then $pos(r) \cap M \neq \emptyset$.

The proof of Claim 2 is similar to that of Claim 1.

Let Q be the subgraph of $DG(P^{(B)})$ induced by M. Q is not an empty graph since $M \neq \emptyset$. Every vertex of Q has an outgoing arc in accordance with Claim 1. By Proposition 6, there is a terminating loop in Q. Consider any terminating loop L of Q. Let R be the external supporting rules for L w.r.t P. For any rule $r \in R$, $pos(r) \cap L = \emptyset$. We show that $r^{(B)} = \emptyset$. Otherwise, since L is a terminating loop and $head(r^{(B)}) \in L$, $pos(r^{(B)}) \cap (M - L) = \emptyset$. Since $pos(r^{(B)}) \cap L = \emptyset$, $pos(r^{(B)}) \cap M = \emptyset$, contradicting Claim 2. Since $r^{(B)} = \emptyset$, either $pos(r) \cap B^- \neq \emptyset$ or $neg(r) \cap B^+ \neq \emptyset$. Hence $BC(r)$ is falsified by B. So the clause $\bigwedge_{(r \in R)} \neg BC(r)$ is true. Clearly $L \cap B^- = \emptyset$. Therefore, the loop formula associated with L is active on B. □

Given a program P and a stable partial assignment B, if ASP_deduce(P, B) returns conflict or implication, the function gen_inferred_clauses, listed in Algorithm 5, computes the active loop formulas by identifying the terminating loops in the subgraph of $DG(P^{(B)})$ induced by M.

An interesting result for tight programs can be inferred from Theorem 7.

Corollary 8. *If a program is tight, neither* implication *nor* conflict *will be re-*

Algorithm 5: Function gen_inferred_clauses

```
gen_inferred_clauses (P, B)
  1 P is a logic program, and B a partial assignment
  2 M ← Atoms(P) − B⁻− Atmost (P, B)
  3 find all terminating loops from the subgraph of DG(P^(B)) induced by M
  4 compute the loop formulas for all terminating loops
  5 return the loop formulas as the inferred clauses.
```

turned by ASP_deduce on a stable partial assignment.

This result can be used to improve the performance of SMODELS on tight programs by eliminating the invocation of the Atmost function.

Theorem 9. *Assume all the SAT functions are correct. SATASP(P) is correct, i.e., it returns an answer set iff there exists an answer set of* P.

Proof. Basically, SATASP(P) enumerates all the possible truth values for all variables of Comp(P). So, SATASP terminates after a finite number of steps.

We then show the following result.

Claim 1. If SATASP returns no answer set, there is no answer set of P; and if it returns a set B of literals, B is an answer set of P.

First, it can be verified that deduce(F∪ lclauses, B) always makes B a stable partial assignment thanks to the unit propagation and F – the completion of P.

Secondly, by Theorem 4 and 7, ASP_deduce(P, B), where P is a program and B a stable partial assignment, returns conflict or implication if and only if there exists a loop formula active on B.

Thirdly, given a program P and a stable partial assignment B, if ASP_deduce(P, B) returns conflict or implication, gen_inferred_clauses(P, B) (Algorithm 5) correctly generates the loop formulas active on B.

If SATASP(P) returns no answer set, there are three cases (Algorithm 3).

Case 1: deduce returns false (line 4). Since deduce is correct, there is no model for F, i.e., $\mathrm{Comp}(P)$. So, there is no answer set of P by Theorem 1.

Case 2: deduce() returns false, and blevel is 0 (line 17). Since deduce, gen_conflict_clauses, and find_blevel are correct, there is no model for $\mathrm{Comp}(P)$ in the *rest* of the search space (the proof of this case is completed later).

Case 3: ASP_deduce(P, B) returns conflict, and blevel is 0 (line 30). Since ASP_deduce returns conflict, there exists a loop formula active on B. The function gen_inferred_clauses correctly generates the active loop formulas leading to the conflict. The variable blevel=0 implies that there is no way to find an assignment in the *rest* of the search space that can satisfy both F and the active formulas.

Now for both case 2 and 3, we show that no partial assignments in the *searched* space can be extended to an answer set of P. In the searched space, a partial assignment B is discarded either because it does not satisfy F (line 12,19) or the active loop formulas generated by gen_inferred_clauses (line 21,27,32). So, none of the partial assignments in the searched space can be extended to an answer set.

Therefore, there is no answer set of P in both case 2 and 3.

If SATASP(P) returns a set B of literals, B is a full assignment (line 40). Hence, B is a model of F, i.e., $\mathrm{Comp}(P)$, because all SAT functions are correct. The full assignment B is obtained either at line 10 or line 12. After that, B has to go through line 21, and ASP_deduce returns nil (otherwise B won't be the assignment returned by SATASP). The value nil implies that there is no loop formula active on B, i.e., all loop formulas are satisfied by B. Therefore B is an answer set of P by Theorem 1.

Since SATASP terminates and returns either no answer set or a set of literals , Claim 1 implies that SATASP is correct. □

CHAPTER 4

SAT-based answer set solving for disjunctive logic programs

In the last chapter, we consider the answer set solving for normal logic programs, i.e., logic programs that allow up to one head atom in each rule. Extending the technology we present there to apply to general disjunctive logic programs, however, is not easy. The main reason is that the complexity of finding an answer set for an arbitrary disjunctive logic program is $\sum_2^P$, which is deemed harder than NP-complete complexity of that for normal logic programs. The extra hardness come from that fact that, given a set of atoms, to test if its an answer set is CO-NP hard.

4.1 Head-cycle-free programs

Some of the disjunctive programs can be transformed to normal logic programs with the same answer sets. Ben-Elyahu and Dechter [2] show this is true for programs with a special syntax property, called *head-cycle-free (HCF)*. A program program is head-cycle-free if it contains no rule that has two different head atoms belonging to the same cycle in the dependency graph of the program.

For a head-cycle-free program P can be transformed into a normal program by replacing each rule r of the form 2.1 by the following rules:

$$\begin{aligned} a_i \leftarrow & a_{k+1}, \ldots, a_m, \text{not } a_{m+1}, \ldots, \text{not } a_n, \\ & \text{not } a_1, \ldots, \text{not } a_{i-1}, \text{not } a_{i+1}, \ldots, \text{not } a_k \end{aligned}$$

for $1 \leq i \leq k$.

Example 4.1.1. *The disjunctive logic program*

$$a \vee b.$$

$$c \leftarrow a.$$

$$d \leftarrow b.$$

is head-cycle-free, and has the same answer sets as the normal program

$$a \leftarrow \text{not } b.$$

$$b \leftarrow \text{not } a.$$

$$c \leftarrow a.$$

$$d \leftarrow b.$$

Example 4.1.2. *The program*

$$a \vee b.$$

$$a \leftarrow b.$$

$$b \leftarrow a.$$

is not head-cycle-free, it has single answer set $\{a, b\}$. *The program*

$$a \leftarrow not\ b.$$

$$b \leftarrow not\ a.$$

$$a \leftarrow b.$$

$$b \leftarrow a.$$

has no answer set.

To compute answer set for head-cycle-free programs, all we need to do is to translate them into normal programs. In the rest of this chapter, we will focus on problem of finding answer set for non head-cycle-free programs.

4.2 Non-head-cycle-free programs

4.2.1 Support formula and active unfounded set

The concept of *unfounded set* for disjunctive logic programs [16] plays a crucial role for the success of the DLV system. In this section we introduce two new concepts derived from it: *support formula* and *active unfounded set.*

Given a logic program P and an assignment B, a set $X \subseteq U(P)$ is an *unfounded set* for P w.r.t. B, if for every atom $a \in X$, for every rule $r \in P$ such that $a \in head(r)$, at least one the the following conditions holds:

1. $pos(r) \cap B^- \neq \emptyset$ or $neg(r) \cap B^+ \neq \emptyset$;
2. $pos(r) \cap X \neq \emptyset$;
3. $(head(r) - X) \cap B^+ \neq \emptyset$.

Given an assignment B for a program P, B is *unfounded-free* if $B \cap X = \emptyset$ for each unfounded set X for P w.r.t. B.

The following proposition shows that unfounded-free property can be used to do answer set testing.

Proposition 10. *[16] Given a model* M *of a logic program* P*,* M *is an answer set of* P *iff* M *is unfounded-free.*

To represent the idea of unfounded set as a propositional expression, we introduce the definition of *support formula.*

Given an assignment B for a logic program P, and a set of atoms $X \subseteq U(P)$. Let

$$R = \{r \mid r \in P, \mathbf{head}(r) \cap X \neq \emptyset, \mathbf{pos}(r) \cap X = \emptyset\}.$$

We called R the *external support* for X. The *support formula* associated with X for P w.r.t. B, denoted by $SF(P, B, X)$ (or just $SF(X)$ when the context is clear), is the following propositional expression:

$$\bigwedge_{r \in R} (\neg BC(r) \vee OH(r)) \supset \bigwedge_{x \in X} \neg x, \tag{4.1}$$

where

$$OH(r) = \bigvee_{b \in \mathbf{head}(r), b \notin X} b.$$

Clearly, when the antecedent (left-hand-side) of the above expression is evaluated as true by the assignment B, X is an unfounded set for P w.r.t. B, which leads to the following result.

Proposition 11. *Given a logic program* P*, a model* M *for* P *is an answer set iff it satisfies every unfounded set formula associated with every set* $X \subseteq U(P)$*.*

Given a program P and an assignment B, $X \subseteq U(P)$ is called an *active unfounded set* if the antecedent of $SF(X)$ is valued as true by B but $X \cap B^- = \emptyset$.

Following Proposition 11, we have

Corollary 12. *Given a program* P *and an assignment* B*, let* X *be an active unfounded set for* P *w.r.t.* B*. If* A *is an answer set extended from* B*, i.e.,* $A \supseteq B^+$ *and* $A \cap B^- = \emptyset$*, then* $A \cap X = \emptyset$*.*

4.2.2 Active loop formula

Given a program P and an assignment B, an active unfounded set X for P w.r.t. B is also called an *active loop* if X forms a cycle in $DG(P)$. We call $SF(X)$ an active loop formula, because it is equivalent to the *loop formula* defined in [14]. In [12] there is a discussion on the connection between loop formula and unfounded sets.

Given a program P and an assignment B, *the current dependency graph*, denoted by $DG(P, B)$, is the directed graph (V, E) such that $V = U(P) - B^-$, and $E = \{(a, b) | r \in P, pos(r) \cap B^- = \emptyset, neg(r) \cap B^+ = \emptyset, a, b \in V, a \in head(r), b \in pos(r)\}$.

A strongly connected component C in a directed graph G is called a *terminating loop* if there is no arc from an vertex in C to any vertex outside it.

An assignment B for program P is said to be *strongly stable*, if for any atom $a \in (U(P) - B^-)$, there is a rule $r \in P$ such that $a \in head(r), pos(r) \cap B^- = \emptyset, neg(r) \cap B^+ = \emptyset, (head(r) - \{a\}) \cap B^+ = \emptyset$.

The following proposition shows the relation between the active unfounded set and the active loop.

Proposition 13. *Given a program* P *and an assignment* B*, Let* X *be an active unfounded set. If* B *is strongly stable, then the subgraph of* $DG(P, B)$ *induced by* X *contains terminating loops, and they are active loops.*

4.2.3 Compute active unfounded set via SAT

Given a logic program P and an assignment B, the problem of finding an active unfounded set can be transformed into a satisfiability problem whose formula is

denote by *AU(P,B)*. *AU(P,B)* is constructed by the following procedure as a set of clauses F:

1. $F = \emptyset$;

2. for every $r \in P$ of the form (2.1), if $head(r) \not\subseteq B^-$ and $pos(r) \cap B^- = \emptyset$ and $neg(r) \cap B^+ = \emptyset$, let $H = head(r) \cap B^+$ and $H = \{h_1, \ldots, h_t\}$.

   (a) If $H \neq \emptyset$,
   $F = F \cup \{(\neg h_1 \vee \cdots \vee \neg h_t \vee a_{k+1} \vee \cdots \vee a_m)\}$.

   (b) If $H = \emptyset$, let $H' = head(r) - B^-$ and $H' = \{h'_1, \ldots, h'_s\}$. For each $i = 1$ to s,
   $F = F \cup \{(\neg h'_i \vee a_{k+1} \vee \cdots \vee a_m)\}$.

3. Let $A = U(P) - B^-$ and $A = \{p_1, \ldots, p_x\}$. $F = F \cup \{(p_1 \vee \cdots \vee p_x)\}$.

Proposition 14. *Given a program* P *and an assignment* B *,* X *is a model for AU(P,B) iff* X *is an active unfounded set of* P *w.r.t.* B.

4.3 SAT-based answer set solving for Non-HCF programs

4.3.1 The algorithm

Based on the results from the previous section, we present a procedure for answer set computation for general (non-HCF) disjunctive logic programs.

It is illustrated by Algorithm 3. The functions *decide*, *deduce*, *analyze_conflicts*, and *backtrack* are the original SAT functions (algorithm 1). The function *encode* translates a program into a propositional formula in CNF form whose models contain all answer sets of the program. The function ASP_deduce looks for an active unfounded set based on the original program and the current assignment. The function *infer_clauses* is used to generate some propositional clauses (called *inferred clauses*) to explain the occurrence of the active unfounded set. If there is an active

Algorithm 6: SAT-based answer set solving for Non-HCF programs

```
//Function: SATASP
// Given: disjunctive logic program P.
F ← encode(P);
blevel ← 0;
while (true) do
    if (decide()≠ done) then
        while (true) do
            if (deduce()=conflict) then
                blevel ← analyze_conflicts();
                if (blevel=0) then
                    return false;
                else
                    backtrack(blevel);
            else
                //Let B be the current assignment.
                X ← ASP_deduce(P,B);
                if (X=∅) then
                    break;
                else
                    IC ← inferred_clauses();
                    F ← F ∪ IC ;
                    if (X ∩ B⁺ ≠ ∅) then
                        blevel ← find_level();
                        if (blevel=0) then
                            return false;
                        else
                            backtrack(blevel)
                    else
                        continue;
    else
        Output answer set;
        return true ;
```

unfounded set X, since all atoms in X cannot be in any answer set extended from the current assignment (Corollary 12), they should be set to $false$. If an atom in X was assigned as true earlier, the system needs to backtrack to a previous state to resolve the conflict. Otherwise, it returns the control to the $deduce$ function.

Note to ensure the correctness of the algorithm, we need to have full executions of the ASP_deduce function upon full assignments. However, as far as partial assignments are concerned, we can always abort the search of an unfounded set after a certain amount of time, and assume such set cannot be found. Practically this is useful, due to the high computational cost of ASP_deduce and since the goal is not about determining the existence of an active unfounded set, but to see if we can find one efficiently to prune the search space.

In the following, we will give detailed discussions on the function $encode$, ASP_deduce, and $infer\_clauses$, followed by an example.

4.3.2 Function $encode$

The input of the function $encode$ is a disjunctive logic program P. Its output is a propositional formula F in CNF form such that $Models(F) \supseteq Ans(P)$ where $Models(F)$ denotes the set of the models for F and $Ans(P)$ the set of the answer sets for P. We allow three different encodings:

1. Simple encoding (SE). One simple implementation of $encode$ is to treat P as a conjunction of propositional rules, i.e., for each rule r of form (2.1) we translate it into the propositional expression $BC(r) \supset HC(r)$, where $HC(r)$ is the expression $a_1 \vee \cdots \vee a_k$.

2. Clark completion [3] encoding (CE). For every atom $a \in U(P)$, We add the

following expression to the SE encoding:

$$a \supset \bigvee_{r \in R} BC(r),$$

where $R = \{r | r \in P, a \in head(r)\}$, is called the *support* for a.

3. LL-completion [14] encoding (LE). For every atom $a \in U(P)$, We add the following expression to the SE encoding:

$$a \supset \bigvee_{r \in R} (BC(r) \wedge EH(r, a)),$$

where R is the support for a and *EH(r, a)* is the expression

$$\bigwedge_{b \in (head(r) - \{a\})} \neg b.$$

Note these formulas need to be transformed into CNF form. It is easy to see that $Models(SE) \supseteq Models(CE) \supseteq Models(LE)$. Even though LE encoding contains the least number of candidate models, as we will show in our empirical studies, its complexity may cause significant performance penalty in many cases.

4.3.3 Function ASP_deduce

The goal of the function ASP_deduce is find to an active unfounded set. When the input program is encoded by the CE or LE encoding, we can directly use the method described in Section 4.2.3. However, in case of SE encoding, we first try to identify a special kind of active unfounded sets, called unsupported sets. Given a program P and an assignment B, for an atom $a \in U(P)$, we say a rule $r \in P$ is a *supporting rule* for a if $a \in head(r)$ and $pos(r) \cap B^- = \emptyset$ adn $neg(r) \cap B^+ = \emptyset$. If $a \notin B^-$ and has no supporting rule, clearly $\{a\}$ is an active unfounded set, called

unsupported set. Note the union of unsupported sets is also an active unfounded set. For CE and LE encoding, the occurrence of unsupported sets is prevented by completion.

Algorithm 7 describes the function ASP_deduce. The function *SAT(AU(P,B),X)* uses a SAT solver to find a model for the formula *AU(P,B)*. It returns *unsatisfiable* if no model exists, and *satisfiable* otherwise. In the latter case, X holds the model. When B is a partial assignment, we impose an execution time limit on the SAT solver, forcing it to return *unsatisfiable* if it can not finish within the time limit.

Algorithm 7: Function ASP_deduce

```
// Given: program P, assignment B
if encoding=SE then
    X ← ∅ ;
    forall a ∈ (U(P) − B⁻) do
        if a has no supporting rule then
            X ← X ∪ {a} ;
    if X ≠ ∅ then
        return X ;
if SAT(AU(P, B),X)= unsatisfiable then
    return ∅;
else
    return X;
```

4.3.4 Function infer_clauses

After the function *ASP_deduce(P,B)* finds an active unfounded set X, we know all atoms in X should be assigned to false. X is usually caused by a subset of B. To support the backjumping and learning mechanism of the SAT solver, we need to encode this information as additional clauses, called *inferred clauses*. Note we don't need to keep all the inferred clauses as they may become irrelevant as when the assignment changes.

For an active unfounded set X, we can use the support formula associated with X, $SF(X)$, as the inferred clauses. If the current assignment is strongly stable, which is always the case when LE encoding is used, the active loop formulas can also be used as the inferred clauses by Proposition 6.

4.3.5 Example

Consider the program P_1 :

$$a \text{ or } b \text{ or } c$$

$$a \leftarrow b, c$$

$$b \leftarrow a, c$$

$$c \leftarrow a$$

$$c \leftarrow b.$$

Its SE encoding $F =$

$$\{(a \vee b \vee c),$$

$$(b, c \supset a),$$

$$(a, c \supset b),$$

$$(a \supset c),$$

$$(b \supset c)\}.$$

The CE encoding is the same as the SE encoding. The LE encoding will add

the following clauses:

$$a \supset ((\neg b \wedge \neg c) \vee (b \wedge c)),$$

$$b \supset ((\neg a \wedge \neg c) \vee (a \wedge c)),$$

$$c \supset ((\neg a \wedge \neg b) \vee a \vee b.$$

Given the assignment $B_1 = \{a, b, c\}$, $AU(P_1, B_1)$ is:

$$\{(\neg a \vee \neg b \vee \neg c),$$

$$(\neg a \vee b \vee c),$$

$$(\neg b \vee a \vee c), (\neg c \vee a),$$

$$(\neg c \vee b), (a \vee b \vee c)\},$$

which has a model $X = \{a, b\}$. X is an active unfounded set for P_1 w.r.t. B_1. The support formula $SF(X) = (c \supset (\neg a \wedge \neg b))$, which is also an active loop formula. the new formula $F \cup \{SF(X)\}$ will have only one model $\{c\}$ which is an answer set for P_1.

CHAPTER 5

Programs with cardinality constraint rules and choice rules

In this chapter we study the answer set solving for logic programs that contain cardinality constraint rules and choice rules. Remember both rules are special cases of weight constraint rules described in section 2.1.4, and any program that contain only cardinality constraints can be transformed into equivalent programs made of constraint rules and choice rules. This transformation is usually taken place during the grounding process.

5.1 Ferraris and Lifschitz's approach

Ferraris and Lifschitz [6] proposed a method to translate cardinality constraint rules and choice rules into another type of extended rules called *nested-expressions*. The nested-expressions are then eliminated by introducing extra atoms. The original program is therefore transformed into a normal program.

5.2 The new approach

We note that the major advantage of using cardinality constraint rules stems from its compactness in expressing some particular cardinality constraint on a set of literals. So the translating approach basically defeats this purpose. The justification is to make possible the black-box use of SAT solver to find answer sets. However, in our case of using SAT solver as a clear-box, we should try to retain the efficiency provided by using choice rules and cardinality constraints as much as possible.

5.2.1 Handle choice rules by extending completion definition

Choice rules can be easily handled by a simple extension of the definition of completion. Given a program P that contains normal rules and choice rules, the

$\mathrm{Comp}(P)$ is defined as:

- For each atom $a \in U(P)$, let R be set of normal rules that have a as head atom, and CR be set of choice rules that have a as one of the head atom,
  - If $R = CR = \emptyset$, $(\neg a)$.
  - Otherwise, $(a \supset (\bigvee_{r \in (R \cup CR)} BC(r)) \wedge (\bigwedge_{r \in R} BC(r) \supset a)$.
- For each rule $r \in P$ such that $head(r) = \emptyset$, $(\neg BC(r))$.

Example 5.2.1. *The completion of program*

$$\{a, b\}.$$
$$a \leftarrow not\ b.$$
$$c \leftarrow not\ a.$$
$$b \leftarrow c.$$

is $(\neg b \supset a)(a \supset (True \vee \neg b))(c \equiv \neg a)(b \supset (c \vee True))(c \supset b)$, *which can be simplified to* $(\neg b \supset a)(c \equiv \neg a)(c \supset b)$. $\mathrm{Comp}(P)$ *has models* $\{a, \neg b, \neg c\}$, $\{a, b, \neg c\}$, *and* $\{\neg a, b, c\}$, *which corresponds to all answer sets of* P.

The definition of dependency graph can be adjusted to handle choice rules by adding edges from head atoms to positive body atoms. It can be verified that all theorems regarding normal programs also apply to normal programs with choice rules after these adjustments.

5.2.2 Handle cardinality constraints by lazy evaluations

Simple cardinality constraint rules can be translated into normal rules without losing much efficiency. For example, the rule $h \leftarrow 1\{a, b, not\ c\}$ can be translated into three rules: $h \leftarrow a$. $h \leftarrow b$. $h \leftarrow not\ c$. For more complex rules like $h \leftarrow$

$50\{a_1, a_2, \ldots, a_{100}\}$, the translation approach will blow up either the number of rules or number of new atoms or both, since there are too many combinations to be considered. Note the last example rule interests us during the answer set search only when the number of true atoms or false atoms among a_1 to a_{100} is around 50. If the number is much less than 50, the rule is unlikely to guide our search; if the number is much higher, the status of the rule is unlikely to change by flipping a few atoms.

The above observation inspires us to develop a lazy evaluation scheme to handle complex cardinality constraint rules. For each complex cardinality constraint in the input program, we introduce a special atom, called *constraint atom*, to replace it in the completion translation. During the answer set search, we periodically check the consistency between the truth values of the constraint atoms and the valuations of the corresponding cardinality constraints. In there are conflicts, the search backtracks and add rules representing the constraint based on the current partial assignments. For example, let $h \leftarrow 2\{a, b, \mathrm{not}\ c, \mathrm{not}\ d\}$ be a constraint rule and C is the constraint atoms for $2\{a, b, \mathrm{not}\ c, \mathrm{not}\ d\}$, partial assignment $\{\neg c, a, b\}$ causes a conflict, which can be resolved by backtracking to a previous assignment level so the rule $(a \wedge b) \supset c$ is not violated.

CHAPTER 6

Empirical studies

6.1 Experimental results on non-disjunctive logic programs

Based on the new procedure described in Chapter 3, we build an answer set solver SAG on top of the state-of-the-art SAT solver MCHAFF[1]. It uses LPARSE [25] to ground the input program which may include cardinality rules and choice rules, which is handled according to the method described in Chapter 5. SAG is tested against SMODELS[2], CMODELS[3] using SAT solvers MCHAFF and ZCHAFF, and ASSAT [4] using MCHAFF on a variety of benchmarks of non-tight programs. We do not include DLV because it is specially designed for disjunctive programs. We do not experiment with tight programs because all SAT-based solvers behave similarly due to the fact that there is a one-to-one correspondence between the answer sets and the completion models.

The experiments are carried out on a DELL Powerage 1850 (two 3.6 GHz Xeon CPUs) with Linux 2.4.21. All solvers use LPARSE to ground the input programs and the time used by LPARSE is counted in the statistics. The systems used are LPARSE-1.0.15, SMODELS-2.28, ASSAT-2.02, CMODELS-3.50 and MCHAFF-spelt3. The results reported are the CPU time in seconds for each solver to find an answer set. In the following tables, we use SM for SMODELS, CM for CMODELS with MCHAFF, CZ for CMODELS with ZCHAFF, P for invocation probability of ASP_deduce, $**$ for time used greater than 600 seconds, $error$ for runtime error, mem for program abort due to insufficient memory.

The first set of programs are those finding Hamiltonian circuit (HC) from the

[1] http://www.printceton.edu/~chaff
[2] http://www.tcs.hut.fi/Software/smodels/
[3] http://www.cs.utexas.edu/users/tag/cmodels.html
[4] http://assat.cs.ust.hk/Assat-2.0/

					SAG	
Graph	SM	CM	CZ	Assat	p=0.5	p=0.1
2xp30.1	0.19	24.72	**	36.15	0.29	0.61
2xp30.2	**	58.65	**	27.46	0.18	1.91
2xp30.3	**	9.84	**	17.18	0.34	0.24
2xp30.4	**	**	463.4	469.6	76.97	73.03
rand2	**	24.2	109.8	93.9	32.69	2.24
rand5	**	3.94	**	40.39	1.51	1.03
rand7	0.49	8.37	**	19.98	1.5	0.87
jbr0.1	**	**	**	**	1.95	2.7
jbr0.2	4.54	**	**	191.6	8.26	5.23
jbr0.3	13.83	**	**	484.1	3.71	8.66
jbr0.4	38.45	**	**	25.95	7.87	4.58
sim7	513.2	14.22	0.87	7.27	0.06	0.06
sim8	**	89.83	**	86.23	0.22	0.23
sim9	**	**	**	168.5	10.97	6.58

Table 6.1: HC problems encoded as normal programs

				SAG	
Graph	SM	CM	CZ	p=0.5	p=0.1
2xp30.1	0.08	35.92	498.88	0.06	0.16
2xp30.2	**	156.25	10.8	0.35	0.65
2xp30.3	**	156.33	10.86	0.39	0.65
2xp30.4	**	**	**	26.47	28.97
rand2	**	192.19	**	12.85	18.66
rand5	**	31.01	**	0.66	6.16
rand7	0.16	398.17	**	0.44	0.26
jbr0.1	0.17	**	**	2.16	0.28
jbr0.2	0.48	51.03	**	0.64	0.54
jbr0.3	0.9	21.1	**	0.24	1.42
jbr0.4	1.57	575.5	**	0.28	0.27
sim7	270.5	4.03	1.01	0.11	0.12
sim8	**	11.07	472.53	0.41	0.98
sim9	**	220.5	**	0.06	0.05

Table 6.2: HC problems encoded as extended programs

				SAG	
BMC	SM	CM	CZ	p=0.5	p=0.1
dp10io2b11	139.51	313.59	35.47	86.9	26.69
dp10so2b8	6.94	13.32	1.04	2.7	1.42
dp12so2b9	120.82	10.84	5	16.39	7.19
dp10io2b12	128.07	17.03	error	2.59	2.52
dp10so2b9	11.54	6.07	4.79	3	1.36
dp12so2b10	308.8	11.53	5.42	3.89	0.88
dp12io2b14	**	78.12	**	75.49	49.21

Table 6.3: Bounded model checking problems

following graphs. Graphs 2xp30.1 – 2xp30.4 are hand-coded graphs[5]; rand2, rand5 and rand7 are random graphs[6]; jbr0.1 – jbr0.4 are random graphs that are generated using ASP benchmark generator JBenge[7], with 80 vertices and the probabilities of the existence of an edge between any two vertices ranging from 0.1 to 0.4; sim7 – sim9 are simplex graphs with levels of 7 – 9 that are also generated by JBenge. Table 6.1 lists the results on the normal program encoding in [20]. Table 6.2 displays the results on the extended program encoding (with cardinality constraints) from http://www.cs.engr.uky.edu/ai/benchmarks.html. ASSAT is not in Table 6.2 since it does not support cardinality constraints.

From Table 6.1 and Table 6.2, we can see that, in most cases, SAG is at least an order of magnitude faster than other systems. In the other cases, it is very close to the top performer. SAG demonstrates its robustness in solving various problem instances. The performance difference of SAG running with the two ASP_deduce invocation probabilities seems to be marginal, compared to the differences between SAG and the other solvers.

The second class of programs are those solving bounded LTL model checking

[5]http://assat.cs.ust.hk/Assat-2.0/hc-2.0.html
[6]http://asparagus.cs.uni-potsdam.de/
[7]http://www.cs.uni-potsdam.de/~konczak/JBenge/index.html

					SAG	
	SM	CM	CZ	Assat	p=0.5	p=0.05
mutex4	16.76	4.6	4.1	4.11	5.44	4.66
phi4	0.21	27.28	error	5.22	0.39	0.34
mutex2	0.32	0.96	0.24	3.56	0.39	0.37
mutex3	146.19	**	error	mem	**	**
phi3	3.57	27.66	4.52	57.94	6.04	4.82

Table 6.4: Checking requirements in a deterministic automaton

problems[8] as described in [10]. They are encoded as extended logic programs. The results are listed in Table 6.3. We can see that SAG running with the *asp_deduce* invocation probability of 0.1 performs better than that with 0.5, and it is the overall winner.

The last set of programs are those solving problems related to checking requirements in a deterministic automaton (http://www.fmi.uni-stuttgart.de/szs/research/projects/synthesis/benchmarks030923.html) [24]. The results are listed in Table 6.4, where problem mutex4 and phi4 are of type "IDFD", and mutex2, mutex3 and phi3 are of type "Morin". As we can see, SAG is comparable to the top performer for all problems except for mutex3 where SMODELS is the only solver that finishes within the time limit of 600 seconds.

[8] http://www.tcs.hut.fi/ kepa/experiments/boundsmodels/

Instance	dlv	gnt2	cmodels	LE	CE	SE
160.1	0.28	0.35	0.5	0.11	0.10	0.09
160.3	0.42	0.05	0.51	0.11	0.12	0.10
75.37	0.29	0.14	0.57	0.51	0.13	0.12
150.2	3.51	2.37	2.10	1.75	1.02	0.10
150.26	1.18	1.45	4.39	3.46	0.18	0.10
125.45	4.96	9.64	**	3.52	10.79	0.23
105.38	8.6	63.2	185	31.87	3.88	3.13
155.0	19.3	44	**	478.64	85.92	0.71
135.11	27.21	34.1	181	176.04	45.85	0.54
155.3	79	97	37.9	53.69	112.0	0.2

Table 6.5: Results on strategic company benchmarks

6.2 Experimental results on disjunctive Non-HCF logic programs

Based on the new procedure described in Chapter 4, we implement a SAT-based answer set solver for disjunctive logic programs, called DSAG. DSAG is built on top of the SAT solver MCHAFF. Its input are programs grounded by LPARSE [25] with disjunctive option. DSAG are tested against DLV[15], GNT2 [11] and CMODELS (using the SAT solver SIMO) on well known benchmarks of disjunctive logic programs. We also provide the comparison of the performances of DSAG with different encodings.

The experiments are carried out on a DELL PowerEdge 1850 (two 3.6 GHz Xeon CPUs) with Linux 2.4.21. The time used for the grounding of the input programs is included in the results. The systems used are DLV(2006-1-12), LPARSE-1.0.15, GNT2 with SMODELS-2.28, CMODELS-3.50 and MCHAFF-spelt3. The results are the CPU time in seconds for each solver to find an answer set or report no answer set exists. The cut off time is 600 seconds, and we use "**" in the tables to represent the fact that a test fails to finish within 600 seconds.

The first problem we test is the strategic company benchmark. It is a $\sum_2^P$-

Instance	dlv	gnt2	cmodels	LE	CE	SE
qbf1	8.20	**	0.05	0.23	0.25	0.68
qbf2	2.47	**	141.0	2.31	2.37	2.45
qbf3	2.20	**	117.0	2.41	2.28	0.85
qbf4	3.16	**	21.3	0.44	0.41	0.40
qbf5	1.33	**	**	5.48	6.31	5.77
qbf6	0.12	**	0.12	0.04	0.04	0.04
qbf7	6.40	**	0.06	0.41	0.44	0.40
qbf8	25.21	**	0.06	0.69	2.25	0.36
qbf9	5.84	1.75	0.05	0.20	0.26	0.24

Table 6.6: Results on 2QBF benchmarks

complete problem. The encoding and the instances of the problem is obtained from the Asparagus[9] system — an online benchmark system for answer set programming. All instances has answer sets. All DSAG instances are so configured that it uses a SAT solver to find active unfounded sets (by invoking $SAT(AU(P, B), X)$) only on full assignments.

The results are listed in Table 6.5 where we use CE, LE, and SE to denote DSAG with the corresponding encodings described in Chapter 4: Clark completion (CE), Lee and Lifschitz completion (LE), and simple encoding (SE). As we can see, DSAG using SE encoding has a clear performance edge over the others. Among three encoding schemes, SE is better than CE which is in turn better than LE. One explaination is that the pruning benefits of more complex encoding does not compensate the higher computational cost caused by the larger formula.

The second problem we test is the Quantified Boolean Formulas (2QBF). It is also a $\sum_2^P$-complete problem. The encoding and the instances of the problem were downloaded from the web-site of Logic and Artificial Intelligence laboratory of the University of Kentucky[10]. In this case, the DSAG solver invoks the function

[9] http://asparagus.cs.uni-potsdam.de/
[10] http://www.cs.uky.edu/ai/benchmark-suite/

$SAT(AU(P, B), X)$ in ASP_deduce on every decision level. The results are presented in Table 6.6. As we can see, the performance is very close among the three DSAG encodings, and overall their performances are solid and competitive.

CHAPTER 7

Conclusions

We have shown how to build efficient SAT-based answer set solvers. The traditional *completion + loop formula* approach is not efficient because it fails to detect the situation when a partial assignment may be extended into a full model, but cannot be extended into any answer set. Our approach adds answer set extensibility checking on partial assignments. When the extensibility checking returns implication or conflict, our approach finds loop formulas active on the current assignment and use them to guide the search. We also apply the similar idea to handle complex cardinality constraint rules. Instead of translate them into large number of clauses with extra variables, we check their consistency during the search and generate individual clauses as needed. For disjunctive programs, we introduce support formula in place of loop formula, and use simple completion encoding in place of complex completion encoding, which also limits the number of clauses and variables for the SAT solver to work on. We have implemented both disjunctive and non-disjunctive answer set solvers on top of the SAT solver MCHAFF. The empirical studies on the well-known benchmarks confirm that the new approach leads to a significant performance boost to the SAT-based answer set solvers.

BIBLIOGRAPHY

[1] Roberto J. Bayardo Jr. and Robert Schrag. Using csp look-back techniques to solve real-world sat instances. In *AAAI/IAAI*, pages 203–208, 1997.

[2] Rachel Ben-Eliyahu and Rina Dechter. Propositional semantics for disjunctive logic programs. *Ann. Math. Artif. Intell.*, 12(1-2):53–87, 1994.

[3] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, NY, 1978.

[4] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, (5):394–397, 1962.

[5] F. Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.

[6] Paolo Ferraris and Vladimir Lifschitz. Weight constraints as nested expressions. *CoRR*, cs.AI/0312045, 2003.

[7] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-driven answer set solving. In *IJCAI*, pages 386–, 2007.

[8] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press.

[9] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Comput.*, 9(3/4):365–386, 1991.

[10] Keijo Heljanko and Ilkka Niemelä. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming*, 3(4&5):519–550, 2003.

[11] Tomi Janhunen, Ilkka Niemelä, Dietmar Seipel, Patrik Simons, and Jia-Huai You. Unfolding partiality and disjunctions in stable model semantics. *CoRR*, cs.AI/0303009, 2003.

[12] Joohyung Lee. A model-theoretic counterpart of loop formulas. In *IJCAI*, pages 503–508, 2005.

[13] Joohyung Lee and Vladimir Lifschitz. Loop formulas for disjunctive logic programs. In Palamidessi [21], pages 451–465.

[14] Joohyung Lee and Vladimir Lifschitz. Loop formulas for disjunctive logic programs. In Palamidessi [21], pages 451–465.

[15] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv system for knowledge representation and reasoning. *CoRR*, cs.AI/0211004, 2002.

[16] Nicola Leone, Pasquale Rullo, and Francesco Scarcello. Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. *Inf. Comput.*, 135(2):69–112, 1997.

[17] Yuliya Lierler and Marco Maratea. Cmodels-2: SAT-based answer set solver enhanced to non-tight programs. In Vladimir Lifschitz and Ilkka Niemelä, editors, *LPNMR*, volume 2923 of *Lecture Notes in Computer Science*, pages 346–350. Springer, 2004.

[18] Fangzhen Lin and Yuting Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. In *AAAI/IAAI*, pages 112–118, 2002.

[19] 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'01)*, 2001.

[20] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intellignece*, 25:241 – 273, 1999.

[21] Catuscia Palamidessi, editor. *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings*, volume 2916 of *Lecture Notes in Computer Science*. Springer, 2003.

[22] João P. Marques Silva and Karem A. Sakallah. Graps: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

[23] Patrik Simons, Ilkka Niemelä, and TImo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.

[24] Alin Stefanescu, Javier Esparza, and Anca Muscholl. Synthesis of distributed algorithms using asynchronous automata. In *CONCUR*, pages 27–41, 2003.

[25] T.Syrianen. Lparse manual, 2003.

[26] Jeffrey Ward and John S. Schlipf. Answer set programming with clause learning. In *LPNMR*, pages 302–313, 2004.

[27] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.