7
Algorithms for Computing Answer Sets

In this chapter we give a short introduction to algorithms for computing answer sets of logic programs. These algorithms form the basis for the implementations of answer set solvers and query-answering systems used in previous chapters. For simplicity we limit ourselves to logic programs without classical negation $\neg$ and constraints (rules with an empty head). This is not a serious restriction because $\neg$ and constraints can always be eliminated from any program Π using Proposition 2.4.4.

The algorithms that we describe can be viewed as typical examples of generate-and-test reasoning algorithms. They have their roots in the Davis-Putnam procedure for finding models of propositional formulas. Understanding this procedure is a stepping-stone to understanding the generate-and-test algorithms implemented in ASP solvers.

7.1 Finding Models of Propositional Formulas

The Davis-Putnam procedure forms the basis of satisfiability solvers; it finds models of propositional formulas by traversing a tree of all possible truth assignments for the variables in that formula. For example, let F be a propositional formula, say

$$F = (X_1 \vee X_2) \wedge \neg X_2.$$

The tree in Figure 7.1 shows all possible assignments of truth values to variables of F. The algorithm traverses the tree looking for a path satisfying F. A simple depth-first search with backtracking allows us to find a path $\langle t, f\rangle$, which is a model of F.

It is clear that the efficiency of this algorithm depends on the ordering of variables and the efficiency of testing if a vector $\bar{X}$ of variables falsifies a propositional formula. For instance, a model of $F \vee (X_3 \vee \neg X_3)$ where F is an arbitrary formula could be found quickly if we were to start by assigning a value to X_3, and if our checking part were smart enough

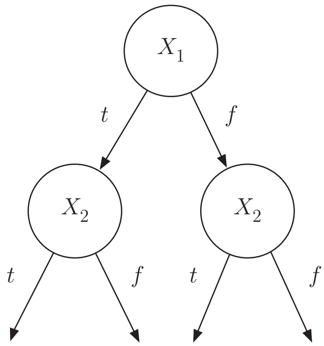

Figure 7.1. All Possible Truth Value Assignments for X_1 and X_2

to recognize that no other variable needs to be examined – an arbitrary assignment of values to these variables would produce a model. Numerous papers address the problem of finding an effective ordering, data structures, and algorithms that would allow for efficient implementations of the Davis-Putnam procedure. Here we describe a simple basic algorithm. Before we go into the details of the algorithm, we define the necessary terminology:

- A *signature* is a set of propositional variables.
- A *clause* over signature Σ is a set $\{l_1, \ldots, l_n\}$ of literals of Σ denoting the disjunction $l_1 \vee \cdots \vee l_n$.
- A *formula* is a set $\{C_1, \ldots, C_m\}$ of clauses denoting the conjunction $C_1 \wedge \cdots \wedge C_m$.[1]
- A *partial interpretation* is a mapping of a set of propositional variables from Σ into truth values. We identify a partial interpretation I with the set of literals made true by I. For any variable p from the domain of I, if $p \in I$ we say that p is *true* in I; if $\neg p \in I$ then p is *false* in I.
- An *interpretation* is a partial interpretation defined on all variables of the language.
- A *model* of a formula F is an interpretation that makes the formula *true*.
- A formula is called *satisfiable* if it has a model.
- Partial interpretation I_2 is *compatible* with partial interpretation I_1 if $I_1 \subseteq I_2$.
- Variable p is *undefined* in partial interpretation I if it does not belong to the domain of I.

[1] Of course, the standard definition of a propositional formula is more general, but the restriction is not overly strong because any such formula can be equivalently written as a conjunction of clauses.

We are interested in the development of an algorithm $Sat(F)$ that takes a formula F as an input and returns a model of F if F is satisfiable and boolean value *false* otherwise. To define the algorithm recursively, we first describe a function $Sat(I, F)$ that searches for a model of formula F over signature Σ compatible with partial interpretation I. To find a model of F we then simply call $Sat(\emptyset, F)$. Here is the algorithm followed by a more detailed explanation.

function Sat
  input: partial interpretation I_0 and formula F_0;
  output: a pair $\langle I, true\rangle$ where I is a model of F_0 compatible with I_0;
    $\langle I_0, false\rangle$ if no such model exists;
var F : formula; I : partial interpretation; X : boolean;
begin
  $F := F_0$;
  $I := I_0$;
  $\langle F, I, X\rangle := Cons(F, I)$;
  if $X = false$ **then**
    return $\langle I_0, false\rangle$;
  if $F = \emptyset$ **then**
    return $\langle I, true\rangle$;
  select variable p undefined in I;
  $\langle I, X\rangle := Sat(I, F \cup \{p\})$;
  if $X = true$ **then**
    return $\langle I, true\rangle$;
  return $Sat(I, F \cup \{\overline{p}\})$;
end;

After initialization of F and I, Sat calls function $Cons$, which computes consequences of F and I, adds these consequences to I, and uses them to simplify F. If no contradiction is derived, the function returns *true* and the updated F and I; otherwise it returns *false* and the original formula and interpretation. (Later we describe a particular implementation of function $Cons$ that is called *unit propagation*.) Next we have two termination conditions. If $Cons$ finds a contradiction, Sat returns *false*. The second condition checks for success. Note that if $F = \emptyset$ then every element of F is vacuously satisfiable and so is F; thus, the function returns *true* together with the new I. It is not difficult to show that I is a desired model of F. If neither condition holds, Sat makes a nondeterministic choice of a yet undefined variable p of F and calls itself recursively.

Now we are ready to define our version of function $Cons$.

function $Cons$ [Unit Propagation];
  input: partial interpretation I_0 and formula F_0 with signature Σ_0;
  output: $\langle F, I, true\rangle$ where I is a partial interpretation such that $I_0 \subseteq I$

and F is a formula over signature Σ_0 such that
M is a model of F_0 compatible with I_0 iff
M is a model of F compatible with I;
$\langle F_0, I_0, false\rangle$ if no such model exists;

var F : formula; I : partial interpretation;
begin
  $F := F_0$;
  $I := I_0$;
  while F contains a unary clause $\{l\}$ **do**
    remove from F all clauses containing l;
    remove from F all occurrences of $\bar{l}$;
    $I := I \cup \{l\}$;
  if $\{\ \} \in F$ **then**
    return $\langle F_0, I_0, false\rangle$;
  return $\langle F, I, true\rangle$;
end;

Note that the condition $\{\ \} \in F$ is used to check if F is already shown to be unsatisfiable. To better understand this condition, note that to satisfy a clause we need to satisfy at least one element in it. Hence, the empty clause is unsatisfiable, and so is the collection F of clauses containing the empty clause. Note also that each call to $Cons$ eliminates occurrences of at least one literal from F, and hence Sat will eventually terminate.

Example 7.1.1. *(Tracing Sat)*
To illustrate the algorithm let us trace the computation of $Sat(I_0, F_0)$ where

$$I_0 = \emptyset$$

and

$$F_0 = \{\{X_1\}, \{\neg X_1, X_2, X_3\}, \{\neg X_1, X_4\}\}$$

First, function $Cons$ goes through two iterations of the loop and returns *true* together with

$$F = \{\{X_2, X_3\}\}$$

and

$$I = \{X_1, X_4\}.$$

It is easy to check that M is a model of F compatible with I iff it is a model of F_0 compatible with I_0. (This, of course, will be true after every iteration of the loop.) The termination conditions in Sat are not satisfied, so the algorithm selects a variable occurring in F and not occurring in I, say X_2, and calls

$$Sat(\{X_1, X_4\}, \{\{X_2, X_3\}, \{X_2\}\}).$$

The new call to $Cons$ returns $I = \{X_1, X_4, X_2\}$ and $F = \emptyset$. Now the second termination condition is satisfied and hence Sat returns $\langle\{X_1, X_4, X_2\}, true\rangle$.

Notice that I is a partial interpretation and hence, strictly speaking, is not a model of the original formula F_0. To make it a model we simply assign arbitrary values to variables that are not in I (in our case, to X_3).

The Sat algorithm is a typical example of the generate-and-test reasoning algorithms used for solving many complex problems in computer science. The practical efficiency of such algorithms depends on several factors including the following:

- The quality of the ordering of variables that determine the selection of an undefined variable in the Sat algorithm: Such orderings are frequently done by *heuristics* – rules of thumb proved to be useful by experience. One can, for instance, use the heuristic that selects the variable and its boolean value that satisfies the maximum number of yet unsatisfied clauses. Sometimes it may be useful to select a variable with the maximum number of occurrences in clauses of a minimum length – doing so increases our chances of arriving at an unsatisfiable clause or of obtaining a unary clause. In many cases the useful heuristics are much more complex and are based on the previous history of computation (e.g., a variable is selected from a clause that has caused the maximum number of conflicts). In all these cases the quality of a heuristic for a particular class of problems is normally determined by extensive experimentation.
- The quality of the procedure computing consequences of a program and a new partial interpretation: The procedure should balance the ability to compute a large number of consequences (which would of course allow a substantial decrease in the search space) and the efficiency of computing these consequences.
- The quality of data structures and the corresponding algorithms used in the actual implementation.

There is a substantial body of knowledge accumulated by computer scientists that allows the designers of solvers to make good choices related to these and other factors. In particular, the designers of answer set solvers have learned a great deal from the research on the design of Sat solvers.

We now describe two algorithms that adapt the basic ideas of Sat to the design of ASP solvers.

7.2 Finding Answer Sets of Logic Programs

Algorithms for computing answer sets of a logic program consist of two steps. In the first step of the computation, the algorithm replaces a program Π, which normally contains variables, by its ground instantiation $ground(\Pi)$. In practical systems $ground(\Pi)$ is not the full set of all syntactically constructible instances of the rules of Π; rather, it is an (often much smaller) subset having precisely the same answer sets as Π. The ability of the grounding procedure to construct small ground instantiation of the program may dramatically affect the performance of the entire system. The grounding techniques implemented by answer set solvers are rather sophisticated. Among other things, they use algorithms from deductive databases and require a good understanding of the relationship among various semantics of logic programming. Nevertheless, the grounding of a program containing variables over large domains can be prohibitively large, which has led to the recent development of answer set solvers that only do partial grounding.

Once the variables have been eliminated from Π, the heart of the computation is then performed by a function we call $Solver$. In this book we present two versions of the function, $Solver1$ and $Solver2$, which work for programs without $\neg$, *or*, and constraints. The first one uses a simple algorithm for computing consequences of the guessing decisions and is readily expandable to disjunctive programs. The second has a more sophisticated computation of consequences and is more efficient but is tailored toward non-disjunctive programs. Even though the structure of $Solver$ is very similar to that of Sat, it works on different objects with sightly different definitions of interpretation, consistency, and so on. Here is the corresponding terminology:

- By *program* we mean a ground logic program without $\neg$, *or*, and constraints.
- By *extended literal*, or simply *e-literal*, over signature Σ we mean a literal of Σ possibly preceded by default negation not. By $not\ l$ we denote $not\ p(\bar{t})$ if $l = p(\bar{t})$ and $p(\bar{t})$ if $l = not\ p(\bar{t})$.
- A set of e-literals is called *consistent* if it contains no e-literals of the form l and $not\ l$.
- A *partial interpretation* of Σ is a consistent set of ground e-literals of Σ. If a partial interpretation I is complete (i.e., for any atom p of Σ either $p \in I$ or $not\ p \in I$), then I is called an *interpretation*.

- An atom p can be *true* ($p \in I$), *false* ($not\ p \in I$), or *undefined* ($p \notin I,\ not\ p \notin I$) with respect to a partial interpretation I.
- An answer set A of a program Π will be represented as an interpretation I of the signature of Π such that

$$I = \{p(\bar{t}) : p(\bar{t}) \in A\} \cup \{not\ p(\bar{t}) : p(\bar{t}) \notin A\}.$$

- A set A of ground atoms is called *compatible* with partial interpretation I if for every ground atom $p(\bar{t})$, if $p(\bar{t}) \in I$ then $p(\bar{t}) \in A$ and if $not\ p(\bar{t}) \in I$ then $p(\bar{t}) \notin A$. A ground program Π is *compatible* with I if it has an answer set compatible with I.

Now we are ready to define our first answer set finding algorithm, $Solver1$.

7.2.1 The First Solver

The Main Program

function $Solver1$
  input: partial interpretation I_0 and program Π_0;
  output: $\langle I, true\rangle$ where I is an answer set of Π_0 compatible with I_0;
      $\langle I_0, false\rangle$ if no such answer set exists;
var Π : program; I : set of e-literals; X : boolean;
begin
  $\Pi := \Pi_0$;
  $I := I_0$;
  $\langle \Pi, I, X\rangle := Cons1(I, \Pi)$;
  if $X = false$ **then**
    return $\langle I_0, false\rangle$;
  if no atom is undefined in I **then**
    if $IsAnswerSet(I, \Pi_0)$ **then**
      return $\langle I, true\rangle$;
    return $\langle I_0, false\rangle$;
  select a ground atom p undefined in I;
  $\langle I, X\rangle := Solver1(I \cup \{p\}, \Pi)$;
  if $X = true$ **then**
    return $\langle I, X\rangle$;
  return $Solver1((I \setminus \{p\}) \cup \{not\ p\}, \Pi)$;
end;

$Solver1$ starts by initializing variables Π and I and calling a function $Cons1(\Pi, I)$, which expands I by a collection of e-literals that can be inferred from Π and I and uses it to simplify Π. If no contradiction is inferred in the process, $Cons1$ returns the new partial interpretation I and the simplified program, together with the boolean value *true*. Otherwise, it returns the original input together with *false*.

The call to $Cons1$ is followed by two termination conditions. First, $Solver1$ checks if $Cons1$ returns *false*. If so, I_0 is inconsistent with Π_0 and $Solver1$ returns *false*. Second, it checks whether partial interpretation I is complete (i.e., if for every atom p either p or $not\ p$ belongs to I). If this is the case then $Solver1$ checks whether I is an answer set. If the answer is *yes* then the function returns *true* together with I. Otherwise, there is no answer set of Π_0 compatible with I_0 and the function returns *false*. If Π still contains some undefined atoms, the function selects such an atom p; $Solver1$ is (recursively) called to explore whether I can be expanded to an answer set of Π containing p. If this is impossible, then $Solver1$ searches for an answer set of Π containing $not\ p$. If one of these calls succeeds, then the function stops and returns *true* together with I. On the failure of both calls, the function returns *false*, because I cannot be expanded to any answer set. As in the case of Sat, an answer set of Π is computed by calling $Solver1(\emptyset, \Pi)$.

Lower Bound – First Refinement of the Consequence Function

Now let us discuss a comparatively simple version of function $Cons1$ traditionally called LB (where LB stands for *lower bound*). A different version of $Cons1$ is discussed later. In what follows we use the traditional name.

Function LB computes the consequences of its parameters, Π and I, using the following four *inference rules*:

(1) If the body of a rule is a subset of I, then its head must be in I.
(2) If atom $p \in I$ belongs to the head of exactly one rule of Π, then the e-literals from the body of this rule must be in I.
(3) If $(not\ p_0) \in I$, $(p_0 \leftarrow B_1, p, B_2) \in \Pi$, and $B_1, B_2 \subseteq I$, then $not\ p$ must be in I.
(4) If Π contains no rule with head p_0, then $not\ p_0$ must be in I.

Let $1 \leq i \leq 3$ be one of the first three inference rules just defined, Π be a program, I be a partial interpretation, and r be a rule of Π. The set of i-consequences of Π, I, and r, denoted by $i\text{-}cons(i, \Pi, I, r)$, is defined as follows: If the *if part* of i is satisfied by Π, I, and r, then $i\text{-}cons(i, \Pi, I, r)$ is the set of e-literals from the i's *then part*. Otherwise, $i\text{-}cons(i, \Pi, I, r) = \emptyset$. (Notice that, for the first inference rule, function $i\text{-}cons$ does not require Π as a parameter, but we nevertheless include it for uniformity of notation.) If $i = 4$ then $i\text{-}cons(i, \Pi, I)$ is the set of literals that do not occur in the heads of rules of Π not falsified by I. Function $LB(I, \Pi)$ can be computed as follows:

function LB
  input: partial interpretation I_0 and program Π_0 with signature Σ_0;
  output: $\langle \Pi, I, true \rangle$ where I is a partial interpretation such that $I_0 \subseteq I$
      and Π is a program with signature Σ_0 such that for every A
      A is an answer set of Π_0 compatible with I_0 iff
      A is an answer set of Π compatible with I;
    $\langle \Pi_0, I_0, false \rangle$ if there is no answer set of Π_0 compatible with I_0;
var I, T : set of e-literals; Π : program;
begin
  $I := I_0$;
  $\Pi := \Pi_0$;
  repeat
    $T := I$;
    remove from Π all the rules whose bodies are falsified by I;
    remove from the bodies of rules of Π all e-literals of the form $not\ l$ satisfied by I;
    select an inference rule i from (1)–(4);
    if $1 \leq i \leq 3$ **then**
      for every $r \in \Pi$ satisfied by the *if part* of inference rule i
        $I := I \cup i\text{-}cons(i, \Pi, I, r)$;
    else
      $I := I \cup i\text{-}cons(4, \Pi, I)$;
  until $I = T$;
  if I is consistent **then**
    return $\langle \Pi, I, true \rangle$;
  return $\langle \Pi_0, I_0, false \rangle$;
end;

First, function LB initializes variables I and Π and simplifies Π by removing from it any rules and e-literals made useless by I. Then LB selects an inference rule (1)–(4) and uses it to expand I by e-literals derived by this rule. The process continues until no more e-literals can be added. If the resulting set I is consistent, LB returns I, the simplified Π, and *true*; otherwise, it returns the original parameters and *false*.

The following example illustrates the function:

Example 7.2.1. *(Tracing LB)*
Consider program P_1:

$$
\begin{aligned}
&p(a) \leftarrow\ not\ q(a).\\
&p(b) \leftarrow\ not\ q(b).\\
&q(a).
\end{aligned}
$$

with the signature determined by the rules of the program, and trace the execution of $LB(\emptyset, P_1)$. Initially I and T are set to $\emptyset$, whereas Π is set to P_1. Application of the two simplifying rules of LB does not change Π. Now the function nondeterministically selects an inference rule. Suppose it selects inference rule (1). The bodies of the first two rules of the program are not satisfied by $\emptyset$. The body of the third rule is empty and hence is satisfied by any set of e-literals, including the empty one. Thus, applying inference rule (1) to the program produces one consequence, $q(a)$, which is added to I.

On the next iteration the simplification deletes the first program rule and hence Π consists of the second and third rule of P_1. Suppose that the inference rule selected next is (4). Since neither $p(a)$ nor $q(b)$ belongs to the heads of the rules of the simplified program, I becomes $\{q(a),\ not\ p(a),\ not\ q(b)\}$.

On the third iteration Π is further simplified to become $\{p(b).\quad q(a).\}$; LB again selects inference rule (1), applies it to the first rule of Π, and sets I to $\{q(a),\ not\ p(a),\ not\ q(b),\ p(b)\}$. Since the next iteration does not produce any new consequences and I is consistent, LB returns $\langle I, \Pi, true\rangle$.

Defining $IsAnswerSet$

Now we discuss a simple algorithm for computing function $IsAnswerSet(I, \Pi)$:

function $IsAnswerSet$
  input: interpretation I and program Π;
  output: *true* if I is an answer set of Π; *false* otherwise;
begin
  Compute the reduct, Π^I of Π with respect to I;
  Compute the answer set, A, of Π^I;
  Check whether $A = atoms(I)$ and return the result;
end;

The first and the last steps of the function are relatively straightforward, but the second one requires some elaboration. We need the following concept: A program Π is called **definite** if Π is a collection of rules of the form

$$p_0 \leftarrow p_1, \ldots, p_n$$

where ps are atoms of signature of Π. (In other words Π contains no default negation.)

Let Π be a definite program and T_Π be an operator defined on sets of atoms from the signature of Π as follows:

$$T_\Pi(A) = \{p_0 :\ p_0 \leftarrow p_1, \ldots, p_n \in \Pi,\ \ p_1, \ldots, p_n \subseteq A\}.$$

Intuitively, $T_\Pi(A)$ returns conclusions of all rules of Π whose bodies are satisfied by A. We use this operator to describe function $Least(\Pi)$, which takes as a parameter a finite definite program Π and returns its answer set. Since by definition of the reduct Π^I is obviously definite, this function can be used to find its answer set.

function $Least$
  input: a definite program Π;
  output: the answer set of Π;
var X, X_0 : set of atoms;
begin
  $X := \emptyset$;
  repeat
    $X_0 := X$;
    $X := T_\Pi(X)$;
  until $X = X_0$;
  return X;
end;

The following example illustrates the computation:

Example 7.2.2. *(Least)*
Consider a program

$$\begin{array}{l} p(a) \leftarrow q(a). \\ q(a). \end{array}$$

Its answer set is obtained as the result of the following computation:

$$T_\Pi(\emptyset) \Rightarrow T_\Pi(\{q(a)\}) \Rightarrow T_\Pi(\{q(a), p(a)\}) \Rightarrow \{q(a), p(a)\}.$$

It may be instructive to check that applying the same algorithm to
$p(a) \leftarrow p(a)$ returns $\emptyset$.

This completes the refinement of $Solver1$. Now let us illustrate how it works by tracing several simple examples. (Remember that LB is just a different name for $Cons1$ from $Solver1$.)

Tracing $Solver1$

Example 7.2.3. *(A Simple Case)*
Consider program P_1 from the previous example:

$$\begin{array}{l} p(a) \leftarrow \ not\ q(a). \\ p(b) \leftarrow \ not\ q(b). \\ q(a). \end{array}$$

To compute the answer set of P_1, we call $Solver1(\emptyset, P_1)$, which starts by initializing Π and I and calling $LB(\emptyset, \Pi)$. As discussed in Example 7.2.1, function LB sets Π to

$$p(b).$$
$$q(a).$$

I to $\{q(a),\ not\ p(a),\ not\ q(b),\ p(b)\}$, and X to *true*. $Solver1$ discovers that, after the first application of LB, all the ground atoms from the signature of P_1 are defined. The last thing needed is to check if I is an answer set of Π. $Solver1$ calls $IsAnswerSet(I, P_1)$, which first computes the reduct P_1^I of P_1 with respect to I. By the definition of reduct we have, that P_1^I is

$$p(b).$$
$$q(a).$$

$Least(P_1^I) = \{q(a), p(b)\}$, which is equal to $atoms(I)$. Hence, $IsAnswerSet(I, P_1)$ returns *true* and the solver returns *true* together with the answer set I of the program.

It is easy to see that program P_1 from Example 7.2.3 is stratified (see Chapter 2) and hence has at most one answer set. The following example illustrates how $Solver1$ works on a program with multiple answer sets.

Example 7.2.4. *(Program with Multiple Answer Sets)*
Now let us compute an answer set of a program P_2:

$$p(a) \leftarrow\ not\ q(a).$$
$$q(a) \leftarrow\ not\ p(a).$$

$Solver1(\emptyset, P_2)$ initializes I and Π and calls $LB(\emptyset, \Pi)$. It is not difficult to see that Π cannot be simplified by $\emptyset$ and that no inference rule used by LB is applicable to the program. Hence LB returns *true* without changing I and Π. $I = \emptyset$, no atom is yet defined, and $Solver1$ starts the selection process. Let us assume that $Solver1$ selects $p(a)$ and makes the recursive call $Solver1(\{p(a)\}, \Pi)$, which, in turn, calls $LB(\{p(a)\}, \Pi)$. After the simplification Π becomes

$$p(a) \leftarrow\ not\ q(a).$$

Suppose that LB selects inference rule (2). It is applicable to the rule of the program, so LB computes a new consequence, $not\ q(a)$; I becomes $\{p(a),\ not\ q(a)\}$ and Π becomes

$$p(a).$$

(Another possibility is to select inference rule (4) instead of (2); this would lead to the same result.) Next, function $IsAnswerSet(\{p(a), not\ q(a)\}, P_2)$ computes P_2^I:

$$p(a).$$

and discovers that $Least(P_2^I) = atoms(I)$; hence, $\{p(a)\}$ is an answer set of P_2. A different choice of a selected e-literal would lead to finding another answer set of the program, $\{q(a)\}$.

Now let us consider a program without answer sets.

Example 7.2.5. *(Detecting Inconsistency)*
Consider a program P_3:

$$p(a) \leftarrow \ not\ p(a).$$

and trace the execution of $Solver1(\emptyset, P_3)$. After the initialization the function calls $LB(\emptyset, \Pi)$. Since no simplification is possible and no inference rule is applicable to Π, $LB(\emptyset, \Pi)$ returns *true* without changing $I = \emptyset$ and $\Pi = P_3$, and $Solver1$ starts its selection process. If $p(a)$ is selected first, then the solver recursively calls $Solver1(\{p(a)\}, \Pi)$. This, in turn, calls $LB(\{p(a)\}, \Pi)$. After the simplification $\Pi = \emptyset$. By inference rule (4) LB concludes $not\ p(a)$, detects inconsistency, and returns *false* together with $\{p(a)\}$. The next call is $Solver1(\{not\ p(a)\}, \Pi)$. This time $LB(\{not\ p(a)\}, \Pi)$ simplifies Π to $p(a)$. Using rule (1) LB obtains inconsistency and $Solver1$ returns *false*. The program has no answer sets.

So far in all our examples $IsAnswerSet$ always returned *true*. In the next example this is not the case.

Example 7.2.6. *(The Importance of $IsAnswerSet$)*
Consider a program P_4:

$$p(a) \leftarrow p(a).$$

and call $Solver1(\emptyset, P_4)$. The program cannot be simplified by $\emptyset$ and none of the four inference rules of LB are applicable to this program, so $LB(\emptyset, \Pi)$ returns *true* without changing I and Π. Now $Solver1$ may select $p(a)$ and call $Solver1(\{p(a)\}, \Pi)$. $LB(\{p(a)\}, \Pi)$ sets I to {p(a)}, does not change Π and returns *true*. All atoms of P_4 are now defined and $Solver1$ calls $IsAnswerSet(\{p(a)\}, P_4)$. The reduct $P_4^I = \Pi$. Obviously, $Least(\Pi) = \emptyset$. Now $IsAnswerSet(\{p(a)\}, P_4)$ compares $\emptyset$ and $\{p(a)\}$, discovers that they are not equal, and returns *false*. $Solver1$ tries another choice, selects $not\ p(a)$, and eventually correctly returns $\{not\ p(a), true\}$.

7.2.2 The Second Solver

In this section we give a different algorithm for computing answer sets of a program. The new algorithm, called $Solver2$, uses a more powerful method for computing consequences than the one used by function LB. In fact, the method is so powerful that checking whether the computed interpretation is indeed an answer set of the program is no longer necessary.

The new consequences-computing function $Cons2$ expands $Cons1$ by computing more negative consequences of the program. (For example, $Cons2$ is able to compute $not\ p$ as a consequence of a program $p(a) \leftarrow p(a)$ with respect to $I = \emptyset$, whereas $Cons1$ cannot.) The computation of these new consequences is done by a new function, $UB(I, \Pi)$, called the **upper bound** of I with respect to Π. Eventually we define $Cons2$ in terms of both functions, LB and UB.

function UB
  input: partial interpretation I_0 and program Π_0 with signature Σ_0;
  output: A set N of e-literals of the form $not\ p$ such that for every A
    A is an answer set of Π_0 compatible with I_0 iff
    A is an answer set of Π_0 compatible with $N \cup I_0$;
var M : partial interpretation; Π : program;
begin
  Let Π be the definite program obtained from Π_0 by
   removing from Π_0 all the rules whose bodies are falsified by I_0 and then
   removing all other occurrences of e-literals of the form $not\ p$;
  $M := Least(\Pi)$;
  $M := \{not\ p : p \in \Sigma_0 \text{ and } p \notin M\}$;
  return M;
end;

Example 7.2.7. *(Upper Bound 1)*
Let us trace $UB(\emptyset, P_4)$ where P_4 is

$$p(a) \leftarrow p(a).$$

First Π is set to P_4. It is easy to see that no simplification of Π by $\emptyset$ is possible and that $Least(\Pi) = \emptyset$. Since the only atom in the signature of Π is $p(a)$, function $UB(\emptyset, \Pi)$ returns $\{not\ p(a)\}$.

Example 7.2.8. *(Upper Bound 2)*
Consider now a program P_5

$$p(a) \leftarrow s(a),\ not\ q(a).$$

This time Π is set to P_5 and $UB(\emptyset, \Pi)$ simplifies Π. Now Π is

$$p(a) \leftarrow s(a).$$

$Least(\Pi)$ returns $\emptyset$. There are three atoms in the signature of P_5: $p(a)$, $q(a)$, and $s(a)$. Hence UB returns $\{not\ p(a), not\ q(a), not\ s(a)\}$.

Now we are ready to define $Cons2$. The function computes consequences of I with respect to Π using both LB and UB.

```
function Cons2
   input: partial interpretation I_0 and program Π_0 with signature Σ_0;
   output: ⟨Π, I, true⟩ where I is a partial interpretation such that I_0 ⊆ I
                and Π is a program with signature Σ_0 such that
                A is an answer set of Π_0 compatible with I_0 iff
                A is an answer set of Π compatible with I;
            ⟨Π_0, I_0, false⟩ if there is no answer set of Π_0 compatible with I_0;
var I, T : set of e-literals; Π : program; X : boolean;
begin
   I := I_0;
   Π := Π_0;
   ⟨Π, I, X⟩ := LB(I, Π);
   if X = true then
      T := UB(I, Π);
      I := I ∪ T;
      if I is consistent then
         return ⟨Π, I, true⟩;
   return ⟨Π_0, I_0, false⟩;
end;
```

Example 7.2.9. *(Cons2 1)*
Consider program P_4 from Example 7.2.7 and trace $Cons2(\emptyset, P_4)$. Recall that P_4 consists of rule

$$p(a) \leftarrow p(a)$$

and that $LB(\emptyset, \Pi)$ where Π is set to P_4 returns $\langle \Pi, \emptyset, true \rangle$. As shown earlier, T is set to $\{not\ p(a)\}$ and the function returns

$$\langle \Pi, \{not\ p(a)\}, true \rangle.$$

Example 7.2.10. *(Cons2 2)*
Consider now a program P_6

$$\begin{array}{l} p(a) \leftarrow s(a), p(a),\ not\ q(a). \\ s(a). \end{array}$$

and trace $Cons2(\emptyset, P_6)$. As usual Π is set to P_6. This time LB returns $\langle \Pi, \{s(a),\ not\ q(a)\}, true\rangle$ where Π consists of the rules

$$\begin{aligned}&p(a) \leftarrow s(a), p(a).\\&s(a).\end{aligned}$$

and has the signature of P_6. UB sets T to $\{not\ p(a), not\ q(a)\}$ and $Cons2$ returns

$$\langle \Pi, \{s(a),\ not\ q(a),\ not\ p(a)\}, true\rangle.$$

Now we can give the new answer set finding algorithm $Solver2$:

function $Solver2$
  input: partial interpretation I_0 and program Π_0;
  output: $\langle I, true\rangle$ where I is an answer set of Π_0 compatible with I_0;
      $\langle I_0, false\rangle$ if no such answer set exists;
var Π : program; I : set of e-literals; X : boolean;
begin
  $\Pi := \Pi_0$;
  $I := I_0$;
  $\langle \Pi, I, X\rangle := Cons2(I, \Pi)$;
  if $X = false$ **then**
    return $\langle I_0, false\rangle$;
  if no atom is undefined in I **then**
    return $\langle I, true\rangle$;
  select a ground atom p undefined in I;
  $\langle I, X\rangle := Solver2(I \cup \{p\}, \Pi)$;
  if $X = true$ **then**
    return $\langle I, X\rangle$;
  return $Solver2(I \cup \{not\ p\}, \Pi)$;
end;

In comparing the efficiency of the two answer set solvers, one can see that $Solver1$ spends less time computing the consequences of the program, but pays for this by spending additional time checking if the computed interpretation is an answer set. $Solver2$ spends more time computing consequences, but does not need the additional checking. Extensive experimentation has shown that the second method of computing this function is usually more efficient. In this case spending more time in computing consequences and avoiding the checking increase the efficiency of the solver. Note, however, that the correctness of the second solver is much less obvious than that of the first. In fact the theorem showing that the interpretation computed by $Solver2$ is an answer set of the program is rather nontrivial.

Another way to improve performance of answer set solvers is to employ a good heuristic for the selection of undefined ground atoms. Selection of a good heuristic is an interesting and important topic for any generate-and-test algorithm. There is a substantial body of research related to the subject. In fact, the area of search and heuristics deserves its own course. In this section we only briefly mention a particular heuristic used in some answer set solvers. The heuristic is rather application-independent and leads to good performance across a range of applications. It can be taken as a starting point for developing more-refined heuristics for particular application areas. First, we replace the selection of an atom by selection of an e-literal. There is no reason to try an atom p first. Sometimes $not\ p$ can do as well or better. Next we try to select an e-literal that has the greatest possibility of changing current partial interpretation I, thereby helping the algorithm discover conflicts or find complete sets of e-literals with a minimal number of choices. For illustrative purposes we give a simple refinement of this idea. First we need a definition. A rule

$$p_0 \leftarrow p_1, \ldots, p_m,\ not\ p_{m+1}, \ldots\ not\ p_n$$

is called *applicable* with respect to a partial interpretation I if

1. $\{p_1, \ldots, p_m\} \subseteq I$,
2. there is no k such that $m + 1 \leq k \leq n$ and $p_k \in I$, and
3. $p_0 \notin I$.

The heuristic *selects an e-literal* $not\ p$ *from the body of an applicable rule with the least number of negated atoms not belonging to* I. For instance, if $I = \{b,\ not\ f\}$ and Π consists of two rules

$$a \leftarrow b,\ not\ c,\ not\ d$$
$$a \leftarrow b,\ not\ e,\ not\ f$$

the heuristic selects $not\ e$. The selection ensures that a is immediately added to I. If we were to select, say, $not\ c$, the expansion of I would have to wait for the next selection of an atom.

7.2.3 Finding Answer Sets of Disjunctive Programs

So far we have only discussed solvers for logic programs not containing disjunction. Dealing with disjunctive programs requires some additional ideas that we briefly discuss in this section. This additional difficulty is not surprising because it follows from the theoretical analysis of the complexity of these two tasks. Finding answer sets of programs not containing epistemic disjunction is an NP-complete problem; therefore, testing (in our case done by $IsAnswerSet$) can be performed in polynomial time (or even eliminated altogether). The problem of computing answer sets for disjunctive programs belongs to a higher complexity class, and hence, checking if a given set of literals is an answer set cannot always be done in polynomial time. (There are, however, large classes of disjunctive logic programs for which the complexity of computing answer sets is NP-complete. Actually *all* of the disjunctive programs we considered so far belong to such a class.) This complexity consideration implies that $Solver2$, which does not check if a computed interpretation is an answer set of a program, cannot be easily adapted to deal with disjunction. $Solver1$, however, is more amenable to change. All we need to do is to replace function $IsAnswerSet$ by a more complex version that works for disjunctive programs. Instead of using a simple computation incorporated in $Least$, we need to check that I satisfies all the rules of the program and, more importantly, that there is no I' such that $atoms(I') \subset atoms(I)$ that also satisfies these rules. This second condition is exactly the one that adds complexity to the algorithm.

7.2.4 Answering Queries

The method for computing answer sets of ASP programs illustrated in the previous section can be used to implement STUDENT-like query-answering systems of ASP. Suppose, for instance, that we are given a consistent program Π and would like to know the answer to a ground query q where q is a literal. The following algorithm allows us to answer this question. (Note that by the call to $Solver$ in this algorithm, we mean $Solver1$ or $Solver2$, whichever implementation is appropriate.)

function $Query$
  input: ground literal l and consistent program Π;
  output: *yes* if l is true in all answer sets of Π,
    no if $\bar{l}$ is true in all answer sets of Π,
    unknown otherwise;
begin
if $Solver(\emptyset, \Pi \cup \{\leftarrow \ not\ l\}) =$ *false* **then**
  return *yes*
if $Solver(\emptyset, \Pi \cup \{\leftarrow \ not\ \bar{l}\}) =$ *false* **then**
  return *no*;
return *unknown*;
end;

Of course the consistency of Π can be checked in advance by a single call to $Solver(\emptyset, \Pi)$.

To answer a query $q_1 \wedge \cdots \wedge q_n$ where qs are ground literals, we expand Π by rules:

$$\begin{aligned} q &\leftarrow q_1, \ldots, q_n \\ \neg q &\leftarrow \neg q_1 \\ &\vdots \\ \neg q &\leftarrow \neg q_n \end{aligned}$$

where q is a new atom. Denote the new program by Π'. Now the question can be answered by a single call to $Query(q, \Pi')$. A similar technique can be used to answer disjunctive query q_1 *or* $\ldots$ *or* q_n. In this case Π should be expanded by rules:

$$\begin{aligned} q &\leftarrow q_1 \\ &\vdots \\ q &\leftarrow q_n \\ \neg q &\leftarrow \neg q_1, \ldots, \neg q_n \end{aligned}$$

Not surprisingly, the approach does not apply to nonground queries. They can be answered by a simple (but not always efficient) algorithm that computes and stores all the answer sets of Π. The analysis of these answer sets allows to return all ground terms t such that query $q(t)$ is true in all the answer sets.

Summary

In this chapter we started by outlining a SAT algorithm for finding models of formulas of propositional logic. This algorithm can be viewed as a typical example of the generate-and-test reasoning algorithm used for solving many complex problems in computer science. We briefly discussed the general structure of such algorithms and several methods for improving their efficiency. Next we showed how the basic ideas of SAT could be adapted to the problem of computing answer sets of nondisjunctive logic programs. In particular we presented two versions of such an algorithm that differ primarily by the functions they use for computing consequences of a program and its new partial interpretation. A short discussion explained how these algorithms can be used to answer simple queries and how the first one could be adapted to work for disjunctive programs. Of course this is only a brief introduction. Serious study of SAT-like methods used to solve problems of non-polynomial complexity requires much more time. Moreover, we are far from fully understanding these algorithms. Many unanswered and fascinating questions remain – after all, many computational problems that need solutions are not polynomial. Discovering methods that allow us to find practical solutions to such problems is crucial for our understanding of computation and for many applications.

References and Further Reading

The Davis-Putnam algorithm for testing satisfiability of propositional formulas was introduced in Davis and Putnam (1960) and further elaborated in Davis, Logemann, and Loveland (1962). Marek (2009) gives a nice introduction and overview of the mathematics of satisfiability. Empirical evaluation of various SAT heuristics can be found, for instance, in Hoos and Stützle (2000). Our presentation of algorithms for computing answer sets of logic programs follows the basic ideas implemented in Smodels and early versions of DLV. See, for instance, Niemela, Simons, and Soininen (2002) and Leone et al. (2006). Recent work establishing close connections between computing answers sets and the SAT algorithms can be found in Lin and Zhao (2004) and Giunchiglia, Lierler, and Maratea (2006). Other approaches investigate computing answer sets via reasoning in difference logic (Janhunen, Niemela, and Sevalnev 2009), mixed integer programming (Liu, Janhunen, and Niemela 2012), and the like. There are also attempts to, at least partially, avoid grounding by integrating ASP-based techniques with that developed in constraint logic programming in Mellarkod, Gelfond, and Zhang (2008), Balduccini (2011), and Ostrowski and Schaub (2012). (See also a solver (Balduccini 2012) that gains efficiency by expanding ASP with non-herbrand functions.) Smart grounders, such as Lparse (Syrjanen 1998), `gringo` (Gebser, Schaub, and Thiele 2007), and DLV's grounder (Alviano et al. 2012) use techniques from deductive databases (Abiteboul, Hull, and Vianu, 1995) and research on logic programming semantics Fitting (1985) and Van Gelder, Ross, and Schlipf (1991). Information on complexity and expressive power of logic programs can be found in Dantsin et al. (2001). A state of the art of design and implementation of ASP solvers can be found in Gebser et al. (2012). Pearl (1984) gives a comprehensive overview of heuristic search theory.

Exercises

1. Given a formula $\{\{a, b, \neg c\}, \{a, \neg b\}, \{\neg b, c, d\}\}$ use the satisfiability algorithm to prove that the formula is satisfiable. *Hint:* Use a reasonable heuristic to order variables of the formula.

2. Given a program

$$\Pi \begin{cases} c \leftarrow a,\ not\ d. \\ a \leftarrow\ not\ b. \\ b \leftarrow\ not\ a. \end{cases}$$

   trace the computation of
   (a) $LB(\{a\}, \Pi)$
   (b) $Solver1(\{a\}, \Pi)$
   (c) $UB(I, \Pi)$ for I returned by LB from (a)
   (d) $Cons2(\{a\}, \Pi)$
   (e) $Solver2(\emptyset, \Pi)$

3. (a) Use Solver1 to compute answer set(s) of program Π shown next. (Assume that a and b are the only constants of the signature of Π.) Do not forget to first modify Π to eliminate classical negation $\neg$ and the constraints.

$$\Pi \begin{cases} p(a) \leftarrow\ not\ \neg p(b). \\ \neg p(b) \leftarrow\ not\ p(a). \\ p(a) \leftarrow\ not\ r(a). \\ r(a) \leftarrow\ not\ p(a). \\ q(X) \leftarrow\ not\ p(X). \\ \neg p(X) \leftarrow\ not\ p(X). \\ \leftarrow r(a). \\ r(b). \end{cases}$$

   (b) How does Π answer queries

$$?q(a) \qquad ?q(b)$$
$$?r(a) \qquad ?r(b)$$

4. Modify $Solver2$ to find *all* answer sets of a given program.
5. Design an algorithm to answer nonground query $Q(X)$.