6
The Answer-Set Programming Paradigm

So far we have used our ASP knowledge bases to get information about the truth or falsity of some statements or to find objects satisfying some simple properties. These types of tasks are normally performed by database systems. Even though the language's ability to express recursive definitions and the methodology of representing defaults and various forms of incomplete information gave us additional power and allowed us to construct rich and elaboration-tolerant knowledge bases, the types of queries essentially remained the same as in databases.

In this chapter we illustrate how significantly different computational problems can be reduced to finding answer sets of logic programs. The method of solving computational problems by reducing them to finding the answer sets of ASP programs is often called the **answer-set programming (ASP) paradigm**. It has been used for finding solutions to a variety of programming tasks, ranging from building decision support systems for the Space Shuttle and computer system configuration to solving problems arising in bio-informatics, zoology, and linguistics. In principle, any NP-complete problem can be solved in this way using programs without disjunction. Even more complex problems can be solved if disjunctive programs are used. In this chapter we illustrate the ASP paradigm by several simple examples. More advanced examples involving larger knowledge representation components are discussed in later chapters.

There are currently several ASP inference engines called *ASP solvers* capable of computing answer sets of programs with millions of ground rules. Normally, an ASP solver starts its computation by grounding the program (i.e., instantiating its variables by ground terms). The resulting program has the same answer sets as the original, but is propositional. The answer sets of the grounded program are then computed using generate-and-test algorithms, which we discuss in Chapter 7.[1]

[1] These algorithms have much in common with classical satisfiability algorithms for propositional logic.

You have, of course, already used an ASP solver if you used STUDENT to run examples or exercises in previous chapters; however, we kept the programs generic enough to run on any solver known to us. Different ASP solvers use various constructs not present in the original ASP discussed in this book. Some of the constructs are aimed at improving the efficiency of the solvers. Others provide useful syntactic sugar that saves programmer's time and may improve the readability of programs. For instance, popular ASP solvers such as **Smodels** and **ClaspD** use program grounders **Lparse** and **Gringo** whose input languages allow so-called choice rules not understood by another popular solver called **DLV**. In turn, DLV allows symbols for lists of terms, and other constructs not understood by Lparse/ Gringo.[2]

There are currently efforts to standardize the input language. Meanwhile, we encourage readers to learn ASP programming by using their favorite ASP system. For standard ASP programs, this book focuses on two systems, `clingo` (Gringo + ClaspD) and DLV, to show the reader some useful features of both input languages. It is likely that, by the time this book is published, the features we mention will have become part of both systems and will be useful regardless of which system you choose. For a quick introduction and tips for using these systems, please see Appendix A. Both systems are good and have many elements that make them special; however, in our wish to keep things simple, we do not discuss all of their unique features.

6.1 Computing Hamiltonian Cycles

We start with describing an ASP solution of finding Hamiltonian cycles of a directed graph. Finding such cycles has applications to numerous important problems, including processor allocation and delivery scheduling. The general problem is stated as follows:

Given a directed graph G and an initial vertex v_0, find a path from v_0 to v_0 that enters each vertex exactly once.

For example, in Figure 6.1, if our initial vertex is a, the Hamiltonian cycle through the graph is a, b, c, d, e, a. Path a, b, c, d, a, e enters every vertex

[2] Gringo was originally based on Lparse, but is evolving. Since Gringo + Clasp D seems to be very efficient and is currently being actively developed and improved, we try to stay compatible with Gringo rather than with Lparse.

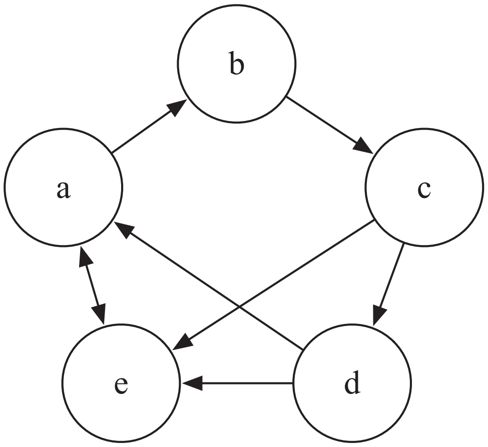


Figure 6.1. Directed Graph

exactly once, but is not a Hamiltonian cycle because it does not end at a. Path a, b, c, d, a is not a Hamiltonian cycle because it misses e.

We now show how the problem of finding a Hamiltonian cycle can be solved using ASP. The main idea is to construct an ASP program, $\Pi_H(G)$, whose answer sets correspond to Hamiltonian cycles of graph G. Once this is done, finding the cycles is reduced to finding answer sets of this program – a task that can be accomplished by an ASP solver. We first give our solution of the Hamiltonian cycle problem in DLV and `clingo`, then give an alternative solution using choice rules of `clingo`.

Not surprisingly, program $\Pi_H(G)$ contains the description of a graph G. The graph is represented by a sort $vertex(V)$ and a collection of atoms of the form $edge(V_1, V_2)$ describing its edges. The initial vertex is specified by atom $init(v_0)$. Note that the information about the input graph is complete, and strictly speaking, this fact should be indicated by the corresponding closed world assumptions. However, negative information about these relations is not relevant to our problem and is omitted for simplicity.

Now we are confronted with the problem of representing Hamiltonian cycles of G. The idea is to represent a cycle by a collection of statements of the form

$$in(v_0, v_1). \; \ldots \; in(v_k, v_0).$$

which belongs to an answer set of $\Pi_H(G)$. (Here $in(V_1, V_2)$ states that "the edge from vertex V_1 to vertex V_2 is in Hamiltonian cycle $\langle v_0, v_1, \ldots, v_k, v_0 \rangle$.")

We start by describing conditions on a collection P of atoms of the form $in(v_1, v_2)$ that will make P a Hamiltonian cycle:

1. P leaves each vertex at most once.
2. P enters each vertex at most once.
3. P enters every vertex of the graph.

The first and the second conditions are encoded by the rules:

```
-in(V,V2) :- in(V,V1),
             vertex(V2),
             V1 != V2.
```

and

```
-in(V2,V) :- in(V1,V),
             vertex(V2),
             V1 != V2.
```

For the third condition, we recursively define relation $reached(V)$, which holds if P enters vertex V on its way from the initial vertex:

```
reached(V2) :- init(V1),
               in(V1,V2).

reached(V2) :- reached(V1),
               in(V1,V2).

-reached(V) :- vertex(V),
               not reached(V).
```

The constraint

```
:- -reached(V).
```

guarantees that every vertex of the graph is entered by our path.

To complete the solution we need to find some way to generate the collection of candidate paths and use these three conditions to select among them those that are Hamiltonian cycles. This can be done by the disjunctive *generation rule*

```
in(V1,V2) | -in(V1,V2) :- edge(V1,V2).
```

that states that every given edge is either in the path or is not in the path. The rule requires our answer sets to contain information about each edge's inclusion in the path. To see this more clearly, let us denote the program consisting of the representation of graph G (edges, vertices, and initial vertex) and the rule on line 16 by Π_0. Notice that there is a one-to-one correspondence between answer sets of Π_0 and arbitrary sets of edges of G. We sometimes say that Π_0 *generates* these sets. Now let Π be Π_0 expanded by the *testing rules* on lines 1–15. This time, there is a one-to-one correspondence between answer sets of Π and Hamiltonian cycles in G. They can be computed by clingo or DLV. Note that we are not really interested in all the information contained in the answer sets of the program – all we need is to display our Hamiltonian cycles (i.e., atoms formed by relation in). To make sure that DLV displays only atoms relevant to describing the Hamiltonian cycle, use option `-filter=in`. You can also use `-pfilter=in` to display only positive atoms formed by relation in. In the input language of `clingo`, display of relevant information is accomplished by a directive, `#show` included in the text of the program. In our case, we simply need to write

```
#show in/2.
```

where 2 refers to the arity of in.

To solve the problem in the example, we need to encode the graph from Figure 6.1 and initial vertex a:

```
17 vertex(a). vertex(b). vertex(c). vertex(d). vertex(e).
18 edge(a,b). edge(b,c). edge(c,d). edge(d,e).
19 edge(e,a). edge(a,e). edge(d,a). edge(c,e).
20 init(a).
```

Let's use `hamgraph.lp` to denote the program on lines 1–20. Calling DLV with

```
dlv -pfilter=in hamgraph.lp
```

we get the following output:

```
{in(a,b), in(b,c), in(c,d), in(d,e), in(e,a)}
```

which represents the only Hamiltonian cycle that starts at a.

There is another solution to the problem that uses a useful extension of the original ASP – the **Choice Rule**. This construct was first introduced in Smodels and is now implemented in all ASP solvers that use Lparse or Gringo as their grounder. Our description is informal; those interested in the formal definition should consult the related literature. A choice rule has two forms:

```
(a)  n1 {p(X) : q(X)} n2 :- body.
(b)  n1 {p(c1);...;p(ck)} n2 :- body.
```

where $n1$ and $n2$ are non-negative integers.[3] In this version of choice rules, both $n1$ and $n2$ can be omitted. Rules of type (a) allow inclusion in the program's answer sets of arbitrary collections S of atoms of the form $p(t)$ such that

1. $n1 \leq |S| \leq n2$
2. If $p(t) \in S$ then $q(t)$ belongs to the corresponding answer set.

If $n1$ is omitted then the above inequality turns into $0 \leq |S| \leq n2$; if $n2$ is omitted, it becomes $n1 \leq |S|$. Note that choice rule (a) can be viewed as a rule defining relation p in terms of previously defined relation q. Rules of type (b) allow selection of such an S from atoms listed in the head of the rule.

Example 6.1.1. *(Choice Rule (a))*
Program

```
q(a).
{p(X) : q(X)}1.
```

has answers sets $\{q(a)\}$ and $\{q(a), p(a)\}$. In the first case, the set S selected by the program is empty; in the second, $S = \{p(a)\}$. Note that both choices satisfy the cardinality constraints.

Example 6.1.2. *(Choice Rule (b))*
Program

```
q(b).
{p(a);p(b)}1.
```

has answers sets $\{q(b)\}$, $\{p(a), q(b)\}$, and $\{p(b), q(b)\}$. Replacing the 1 in the last rule by a 2 gives the original three answer sets, plus one more – $\{p(a), p(b), q(b)\}$.

Using the first form of the choice rule allows us to replace the disjunctive generation rule from line 16 by

[3] Actually `clingo` allows more general rules of the form
(a) `n1 OP1 {p(X) : q(X)} OP2 n2 :- body.`
(b) `n1 OP1 {p(c1);...;p(ck)} OP2 n2 :- body.`
where the OPs are relations <, >, =, !=, <=, or >=, but we will not use it in this book.

```
{in(V1,V2) : edge(V1,V2)}.
-in(V1,V2) :- vertex(V1), vertex(V2),
              not in(V1,V2).
```

The first is the generation rule. (Note that the second rule is not really necessary for finding the solution to our puzzle and can be omitted.) To make sure that only relevant information is displayed we, of course, need to also include `#show in/2`.

You just saw your first example of a successful application of ASP methodology to solving a classical combinatorial problem. Now it may be prudent to spend some time reflecting on this experience. As you know, one of the main goals of computer science is to discover new ways of solving computational problems. (Think of the impact the discovery of recursion had on our ability to do that!) From this perspective, it is instructive to compare the *processes* of finding "procedural" versus "declarative" solutions to the Hamiltonian cycle problem. They are markedly different and lead to markedly different implementations. The first focuses on data structure and algorithm; the second on the appropriate encoding of the definition of the problem. We strongly encourage you to spend some time finding and implementing the procedural solution. But even without this exercise, one can probably see that the declarative solution is shorter, easier to implement (at least for those who had mastered both methodologies), more transparent, and more reliable. An important open question is "What are the limits of applicability of the second method?" (Perhaps some of you may decide to contribute to finding the answer to this question.)

Let us also mention that another declarative solution to the problem of finding Hamiltonian cycles had been discovered before ASP was even developed. In this solution a graph G and the definition of Hamiltonian cycle were encoded by a propositional formula F. There is a one-to-one correspondence between models of F and Hamiltonian cycles of G. A program, called a **satisfiability solver** finds the models. Computer scientists have been developing satisfiability solvers for propositional logic for more than 50 years and succeeded in producing remarkably efficient systems. So, why use ASP? There are two reasons to do so.

1. Often the ASP encoding is much shorter and easier to understand. Some recent mathematical results show that this feature is not an accident: Any equivalent translation from logic programs to propositional formulas involves a significant increase in size.
2. There are complexity results that prove that ASP with disjunction has more expressive power than propositional logic. Some problems

that can be solved by disjunctive ASP simply cannot be solved by satisfiability solvers.

The attractiveness of ASP does not mean that all the remarkable work on satisfiability solvers is in danger of becoming useless. The advantages and disadvantages of both methods are still under investigation. But more importantly the developers of ASP solvers are rapidly finding ways to use ideas from satisfiability theory, as well as actual, off-the-shelf satisfiability solvers, to build new and more efficient answer set solvers.

6.2 Solving Puzzles

The ability to solve puzzles is an important part of human intelligence. The skill with which students are able to do this is often used in decisions of whether to admit them to graduate school, offer them a job, and so on. Even though reasonable people can question the wisdom of such policies, few would argue that the ability to solve puzzles is at least one measure of intelligence. In this section we use ASP to design programs for solving several interesting puzzles.

6.2.1 Sudoku Puzzle

We start with a popular Japanese puzzle game called Sudoku. Our solution simply represents the Sudoku rules in ASP. The answer sets of the resulting program correspond to solutions of the puzzle.

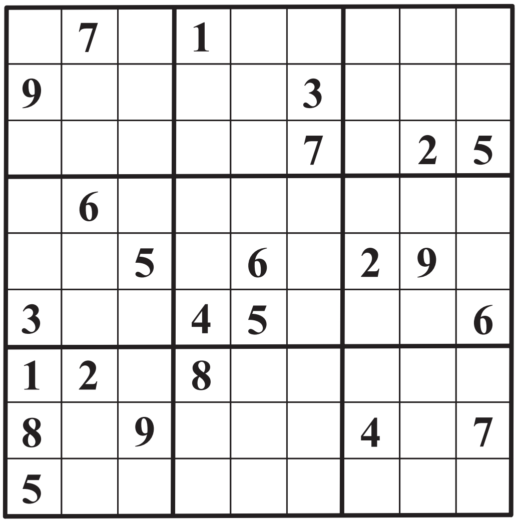
Figure 6.2 shows a typical puzzle. As you can see, the game is played on a 9×9 grid that is further subdivided into nine 3×3 regions. Initially the grid contains numbers in some of its locations. A player must place the numbers 1 through 9 in the grid so that the following conditions are satisfied:

1. Each location contains a single number.
2. No row contains the same number twice.
3. No column contains the same number twice.
4. No 3×3 region contains the same number twice.

(Typically, for a given initial situation, the puzzle has exactly one solution.)

To describe the Sudoku puzzle's domain, we need names for the grid's locations and regions. We use coordinates – pairs of numbers from 1 to 9 – to name locations. Regions are numbered from 1 to 9. To describe numbers placed in the grid's locations, we use relation

$$pos(N, X, Y) \text{ – “number } N \text{ is placed in location } (X, Y).\text{”}$$



	7		1					
9					3			
					7		2	5
	6							
		5		6		2	9	
3			4	5				6
1	2		8					
8		9				4		7
5								

Figure 6.2. Sudoku Puzzle

We also need a relation

$$in\_region(X, Y, R) \text{ – “location } (X, Y) \text{ belongs to region } R\text{.”}$$

We are looking for a collection of atoms of the form $pos(N, X, Y)$ that satisfies the four conditions given earlier. For every coordinate (X, Y) there is a single number N such that $pos(N, X, Y)$ belongs to this collection. Using disjunction this can be expressed as follows:

```
1 num(1..9).
2 coord(X,Y) :- num(X), num(Y).
3 pos(1,X,Y) | pos(2,X,Y) | pos(3,X,Y) |
4 pos(4,X,Y)  | pos(5,X,Y) | pos(6,X,Y) |
5 pos(7,X,Y) | pos(8,X,Y) | pos(9,X,Y)  :-coord(X,Y).
```

The rule on lines 3–5 generates all possible assignments of numbers to locations. These assignments are tested against the remaining constraints by the following rules:
No row contains the same number twice:

```
6 -pos(N,X,Y2) :- pos(N,X,Y1),
7                 coord(X,Y2),
8                 Y1 != Y2.
```

No column contains the same number twice:

```
 9 -pos(N,X2,Y) :- pos(N,X1,Y),
10                 coord(X2,Y),
11                 X1 != X2.
```

No region contains the same number twice:

```
12 -pos(N,X2,Y2) :- pos(N,X1,Y1),
13                   in_region(X1,Y1,R),
14                   in_region(X2,Y2,R),
15                   X1 != X2,
16                   Y1 != Y2.
```

Last, we define relation in_region. Given coordinates X and Y, their region R can be computed by the following formula:

$$R = ((X-1)/3) * 3 + ((Y+2)/3).$$

It is encoded in DLV by the following rule:

```
17 in_region(X,Y,R) :- num(X), num(Y), num(Z1), num(Z2),
18                      num(Z3), num(Z4), num(Z5), num(R),
19                      Z1 = X-1,
20                      Z2 = Z1/3,
21                      Z3 = Z2*3,
22                      Z4 = Y+2,
23                      Z5 = Z4/3,
24                      R = Z3 + Z5.
```

We can represent an initial position in the grid from Figure 6.2 by a collection of atoms of the form $pos(n, x, y)$, and compute answer sets of this program. These contain the solutions of the puzzle. For example, the puzzle in Figure 6.2 can be encoded as follows:

```
25 pos(7,2,1).  pos(1,4,1).  pos(9,1,2).  pos(3,6,2).
26 pos(7,6,3).  pos(2,8,3).  pos(5,9,3).  pos(6,2,4).
27 pos(5,3,5).  pos(6,5,5).  pos(2,7,5).  pos(9,8,5).
28 pos(3,1,6).  pos(4,4,6).  pos(5,5,6).  pos(6,9,6).
29 pos(1,1,7).  pos(2,2,7).  pos(8,4,7).  pos(8,1,8).
30 pos(9,3,8).  pos(4,7,8).  pos(7,9,8).  pos(5,1,9).
```

Let's call the program consisting of lines 1–30 `sudokudlv.lp`. Invoking DLV with

```
dlv -pfilter=pos sudokudlv.lp
```

produces the following output:

```
{pos(1,1,7), pos(1,4,1), pos(2,2,7),
 pos(2,7,5), pos(2,8,3), pos(3,1,6),
 pos(3,6,2), pos(4,4,6), pos(4,7,8),
 pos(5,1,9), pos(5,3,5), pos(5,5,6),
 pos(5,9,3), pos(6,2,4), pos(6,5,5),
 pos(6,9,6), pos(7,2,1), pos(7,6,3),
 pos(7,9,8), pos(8,1,8), pos(8,4,7),
 pos(9,1,2), pos(9,3,8), pos(9,8,5),
 pos(6,1,1), pos(4,1,3), pos(2,1,4),
 pos(7,1,5), pos(5,2,2), pos(1,2,3),
 pos(8,2,5), pos(9,2,6), pos(3,2,8),
 pos(4,2,9), pos(2,3,1), pos(8,3,2),
 pos(3,3,3), pos(4,3,4), pos(1,3,6),
 pos(7,3,7), pos(6,3,9), pos(2,4,2),
 pos(6,4,3), pos(9,4,4), pos(3,4,5),
 pos(5,4,8), pos(7,4,9), pos(9,5,1),
 pos(4,5,2), pos(8,5,3), pos(7,5,4),
 pos(3,5,7), pos(2,5,8), pos(1,5,9),
 pos(5,6,1), pos(8,6,4), pos(1,6,5),
 pos(2,6,6), pos(4,6,7), pos(6,6,8),
 pos(9,6,9), pos(3,7,1), pos(6,7,2),
 pos(9,7,3), pos(1,7,4), pos(7,7,6),
 pos(5,7,7), pos(8,7,9), pos(4,8,1),
 pos(7,8,2), pos(5,8,4), pos(8,8,6),
 pos(6,8,7), pos(1,8,8), pos(3,8,9),
 pos(8,9,1), pos(1,9,2), pos(3,9,4),
 pos(4,9,5), pos(9,9,7), pos(2,9,9)}
```

(If you wish to check the answer, note that the first four rows contain the input information, whereas the rest contain the solution numbers in a reasonable order.) Our solution can be viewed as a typical example of answer-set programming methodology.

The same problem can, of course, be solved using the choice rules of `clingo`. To do that we simply replace the generating disjunctive rule on lines 3–5 by the following choice rule:

```
1{pos(N,X,Y):num(N)}1 :- coord(X,Y).
```

Clingo allows us to simplify the rule from lines 17–24 by

```
in_region(X,Y,((X-1)/3)*3+((Y+2)/3)) :- num(X), num(Y).
```

For display purposes, we recommend inserting

```
#show pos/3.
```

6.2.2 Mystery Puzzle

A detective or mystery story can also be an interesting challenge for a rational agent. Unlike in Sudoku, it is not enough to know the puzzle's "rules" to find its solution. One normally also needs to make some assumptions that come from our general knowledge about the world. It means that programs solving such puzzles should possess some commonsense knowledge. Let us look at an example.

Vinny has been murdered, and Andy, Ben, and Cole are suspects. Andy says he did not do it. He says that Ben was the victim's friend but that Cole hated the victim. Ben says he was out of town the day of the murder, and besides he didn't even know the guy. Cole says he is innocent and he saw Andy and Ben with the victim just before the murder. Assuming that everyone – except possibly for the murderer – is telling the truth, use ASP to solve the case.

The story is about four people:

```
person(andy). person(ben). person(cole). person(vinny).
```

The next several statements record their testimony. Relation $says(P, S, 1)$ holds if person P says that statement S is true; $says(P, S, 0)$ holds if P says that S is false. The corresponding statements are represented by self-explanatory terms (e.g., $murderer(andy)$, $friends(ben, vinny)$, etc).

```
%% Andy says:
says(andy, murderer(andy), 0).     %% He didn't do it.
says(andy, hated(cole,vinny), 1).  %% Cole hated Vinny.
says(andy, friends(ben,vinny), 1). %% Ben and Vinny
                                   %% were friends.
%% Ben says:
says(ben, out_of_town(ben), 1).  %% He was out of town.
says(ben, know(ben,vinny), 0). %% He didn't know Vinny.
%% Cole says:
says(cole, innocent(cole), 1).       %% He is innocent.
says(cole, together(andy,vinny), 1). %% He saw Andy and
says(cole, together(ben,vinny), 1). %% Ben with victim.
```

The next two rules formalize the last statement of the puzzle, using relation $holds(F)$ – “statement F is true”: Everyone, except possibly for the murderer, is telling the truth:

```
holds(S) :- says(P,S,1),
            -holds(murderer(P)).
-holds(S) :- says(P,S,0),
             -holds(murderer(P)).
```

The next rule states that one of the suspects is a murderer:

```
holds(murderer(andy)) | holds(murderer(ben)) |
holds(murderer(cole)).
```

The next set of rules encodes some commonsense knowledge about the meaning of the relations used by the suspects. We start with proclaiming our belief that normally people are not murderers:

```
-holds(murderer(P)) :- person(P),
                       not holds(murderer(P)).
```

Next we specify that some relations are symmetric and/or transitive: Relation *together* is symmetric and transitive:

```
holds(together(A,B)) :- holds(together(B,A)).
holds(together(A,B)) :- holds(together(A,C)),
                        holds(together(C,B)).
```

Relation *friends* is symmetric:

```
holds(friends(A,B)) :- holds(friends(B,A)).
```

Several other properties express the mutual exclusivity of some of the relations mentioned in the story. Since these conditions are not used to *define* the corresponding relations, but rather relate two concepts to each other, they are represented by constraints.
Murderers are not innocent:

```
:- holds(innocent(P)),
   holds(murderer(P)).
```

A person cannot be seen together with people who are out of town:

```
:- holds(out_of_town(A)),
   holds(together(A,B)).
```

Friends know each other:

```
:- -holds(know(A,B)),
   holds(friends(A,B)).
```

A person who was out of town cannot be the murderer:

```
:- holds(murderer(P)),
   holds(out_of_town(P)).
```

To display the answer we introduce relation $murderer$ defined as follows:

```
murderer(P) :- holds(murderer(P)).
```

The only answer set of this program contains $murderer(ben)$, correctly concluding that Ben is the murderer. Clearly, enough commonsense knowledge was added to get the unique answer. Actually not all of this knowledge is even necessary – some of the constraints can be dropped without influencing the result. (We advise the reader to see which constraints can be safely eliminated.)

To rewrite the program for use with choice rules of `clingo`; we replace line 18 by the following choice rule:

```
1{holds(murderer(andy));holds(murderer(ben));
holds(murderer(cole))}1.
```

The rest of the program remains the same.

Summary

In this chapter we discussed the method of solving a computational problem P by writing an ASP program Π_P whose answer sets correspond to the problem's solutions and then using ASP solvers to find those solutions. Π_P normally consists of several parts: a knowledge base containing general knowledge related to the problem, a description of a particular instance of the problem, and a generator of possible solutions. Informally the last element is used to generate candidates for the solutions of the problem instance, whereas the first two parts are used to check if a candidate is indeed a solution. The general knowledge about the Hamiltonian cycles problem from Section 6.1 is recorded by the rules on lines 1–15. The candidate solution generation rule is given on line 16 (or by its choice rule alternative). The problem instance corresponding to Figure 6.1 is given on lines 17–20. It may be interesting to note that, in the Hamiltonian cycle example, relevant knowledge takes the form of the definition of Hamiltonian cycle and is completely separate from generation.

In the Sudoku examples the definition of a solution of the Sudoku puzzle consists of rules on lines 1–24. Note that it includes the generating rule (which says that such a solution must be a function assigning numbers to coordinates). The specific instance is given on lines 25–30.

In the mystery puzzle the precise mathematical definition of the solution does not exist. Instead the rules on lines 19–32 contain commonsense knowledge about the meaning of terms used in the story. As was mentioned earlier, these rules are not the definitions of the corresponding relations. They only contain partial knowledge about those relations represented by logic programming constraints. As a result a lot of negative information about them is missing. Fortunately, this lack of knowledge does not prevent us from correctly solving the problem.

The examples discussed in this chapter covered a number of different computational tasks. In the next few chapters, we use the same techniques to solve several classical AI problems including those related to planning and diagnostics. First, however, we present the basic algorithms for computing answer sets of a logic program.

References and Further Reading

Answer set programming as the method of solving nontrivial search problems was first advocated in Marek and Truszczynski (1999) and Niemela (1999). The method only became possible because of the development of a number of efficient answer set solvers Simons (1996), Smodels Web Page, Gebser et al. (n.d.), DLV Web Page, and Lierler and Marateo (n.d.). *Answer Set Solving in Practice* (Gebser et al. 2012) is a great introduction to practical applications of ASP with emphasis on efficiency and multiple advanced features of ASP languages not covered in our book. Results comparing the expressive power of Answer Set Prolog and propositional logic can be found in Lifschitz and Razborov (2006). A system solving puzzles formulated in natural language by translating them into formal ASP problems is described in Baral and Dzifcak (2012). Another ASP based "puzzle solver" that uses its own input language for formulating puzzles can be found in Truszczynski, Marek, and Finkel (2006).

Exercises

1. Run the Hamiltonian cycle program on graphs represented by the following statements. Give the answer set(s).

   (a)
   ```
   vertex(a).  vertex(b).  vertex(c).
   edge(a,b).  edge(b,c).  edge(c,a).
   init(a).
   ```

   (b) Same as exercise (1a) except replace `edge(c,a)` by `edge(a,c)`.

   (c)
   ```
   vertex(a).  vertex(b).  vertex(c).  vertex(d).
   edge(a,b).  edge(b,c).  edge(c,d).  edge(d,a).
   edge(c,a).  edge(a,a).
   init(a).
   ```

   (d) Same as exercise (1) except add

   ```
   edge(b,a).  edge(c,b).  edge(d,c).  edge(a,d).
   ```

2. Give the answer sets for the following program:

   ```
   q(a).
   q(b).
   1{p(X):q(X)}2.
   ```

3. Give the answer sets for the following program:

   ```
   {p(a);p(b)}2.
   ```

4. Suppose we wanted to separate the definition of the Sudoku problem rules from the generating part (lines 1–5). We could replace it by

   ```
   num(1..9).
   coord(X,Y) :- num(X), num(Y).
   pos(I,X,Y) | -pos(I,X,Y) :- num(I), coord(X,Y).
   ```

   What rules should be added to the program to complete the solution of the Sudoku problem? *Hint:* Encode the information necessary to describe the puzzle requirement that every location must be filled in with a unique number.

5. In the mystery puzzle, it is debatable whether the rule on lines 31–32 about murderers being in town is a valid assumption. Is it necessary to solve the crime?

6. Use a solver to find the answer set for the mystery program without suppressing the output of other predicates.

(a) Does the program correctly reject the rest of Ben's testimony?

(b) What would happen if anyone could be lying? Remove the rules on lines 14–17 and test whether the program conforms to your intuition.

(c) What if we were to assume that friends do not murder each other? For simplicity, just use rule

```
:- holds(friends(P,vinny)), murderer(P).
```

which states that a friend of Vinny's would not murder him. What happens when you add this rule to the original program? Why?

7. Given a round table with ten chairs and a group of ten people, some of whom are married and some of whom do not like each other, use ASP to find a seating assignment for members of this group such that husbands and wives are seated next to each other and no neighbors dislike each other.