Answer set programming (asp)

So far we used our A-Prolog knowledge bases to receive information about 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' ability to express recursive definitions and the methodology of representing defaults and various forms of incomplete information gave us additional power and allowed to construct rich and elaboration tolerant knowledge bases the type of queries essentially remained the same as in databases. In this section we will illustarte how significantly different computational problems can be reduced to finding answer sets of logic programs.

Computing Hamiltonian Paths

Given: Directed graph G, initial vertex s_0.

Find: a path from s_0 to s_0 which visit each vertex exactly once.

Graph represented by

$vertex(s_0)\ldots.$ $edge(s_i, s_j)\ldots.$ $init(s_0).$

The Idea: Represent every Hamiltonian path by a collection of statements of the form

$in(s_0, s_1).$ $\ldots in(s_k, s_0).$

which belongs to an answer set of a program Π associated with the problem.

Constructing the program

We start with describing conditions on a collection P of atoms of the form $in(s1, s2)$ which will make P a Hamiltonian path.

- **Path P visits every vertex V at most once:**

```
(1)    :- vertex(V2), vertex(V1), vertex(V),
          in(V1,V),
          in(V2,V),
          neq(V1,V2).
(2)    :- vertex(V2), vertex(V1), vertex(V),
          in(V,V1),
          in(V,V2),
          neq(V1,V2).
```

- *P* **visits every vertex of the graph.**

First we introduce relation *$reached(V)$* **which holds if** *P* **visits** *V* **on its way from the initial vertex:**

```
(3)  reached(V2) :- vertex(V1), vertex(V),
                    init(V1),
                    in(V1,V2).
(4)  reached(V2) :- vertex(V1), vertex(V),
                    reached(V_1),
                    in(V1,V2).
```

Now the constraint

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

guarantees that every vertex is reached.

To complete the solution we need to find some way to generate the collection of 'candidate' paths and use the above conditions to select those which are indeed Hamiltonian. In *DLV* this can be done by the rule

```
(6)    in(V1,V2) v -in(V1,V2) :- edge(V1,V2).
```

Let us denote the program consisting of the representation of the graph G and this rule by Π_0. Obviously there is one-to-one correspondence between answer sets of Π_0 and arbitrary sets of edges of G. We will some times say that Π_0 generates these sets. Now let Π be Π_0 expanded by 'testing' rules above. This time, there is one-to-one correspondence between answer sets of Π and Hamiltonian paths in G. They can be computed by *DLV*.

The problem can also be solved in SMODELS. To do that we will replace our disjunctive rule (6) by two non-disjunctive rules

```
7a.  in(V1,V2)  :- edge(V1,V2),
                   not out(V1,V2).
7b.  out(V1,V2) :- edge(V1,V2),
                   not in(V1,V2).
```

Assuming that *out* **is just an another name for negation of** *in* **it is easy to check that both programs have the same answer sets.**

Another useful construct of SMODELS, called a Choice Rule, has the form:

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

Both, $n1$ and $n2$ can be omitted. The rule (a) allows to include in answer sets of the program arbitrary collections S of atoms of the form $p(t)$ such that

1. $n1 \leq |S| \leq n2$

2. $p(t) \in S$ **then** $q(t)$ **belongs to the corresponding answer set.**

The rule (b) allows to select such an S from atoms listed in the head of the rule. E.g., program

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

has answers sets $\{q(a)\}$ **and** $\{q(a), p(a)\}$**.**

The rules (7) can be replaced by the rule

```
{in(V1,V2) : edge(V1,V2)}.
```

Comments

• One of the goals of CS is to discover new ways of solving computational problems. (Think about the impact the discovery of recursion had on our ability to solve problems.)

From this perspective it is instructive to compare the PROCESSES of finding a "procedural" and a "declarative" solutions of the above problem. They are markedly different and lead to different solutions. The first focuses on data structure and algorithm. The second on the appropriate encoding of the definition of the problem.

Question: What are the limits of applicability of the second method?

Comments

Other declarative solution of the problem of finding Hamiltonian paths was tried before. A graph, G, and the definition of Hamiltonian path can be encoded by a propositional formula F. There is one-to-one correspondence between models of F and Hamiltonian paths of G. A program, called "satisfiability checker" finds the models.

So, why use A-Prolog?

- **A-Prolog encoding is much shorter and easier to understand. Seems to be frequently the case. Need mathematical analysis!**
- **A-Prolog with disjunction has more expressive power then propositional logic.**

Solving Puzzles

Consider the following puzzle:

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

Write a program to solve the puzzle.

The story is about four people: person(a). person(b). person(c). person(v). Next several statements record their testimony.

```
Arthur says:
says(a,i(a)).      he is innocent.
says(a,ht(c,v)).   Carleton hated Viktor.
says(a,f(b,v)).    Bertram and Victor are
                   friends.
Bertram says:
says(b,o(b)).      he was out of town.
says(b,nk(b,v)).   he didn't know Viktor.


Carleton says:
says(c,i(c)).       he is innocent.
says(c,t(a,v)).     he saw Arthur and Bertram
says(c,t(b,v)).     with the victim.
```

The rule

```
h(F) :- says(P,F),
        not m(P).
```

where $h(F)$ reads 'holds F', says that everyone, except possibly for the murderer, is telling the truth. Next several rules contain some commonsense knowledge about meaning of the relations used by the suspects.

1. Relation 'together' is symmetric and transitive:

```
h(t(A,B)) :- person(A),person(B),
             h(t(B,A)).
h(t(A,B)) :- person(A),person(B),person(C),
             h(t(A,C)),
             h(t(C,B)).
```

2. Relation ’friends’ is symmetric:

```
h(f(A,B)) :- person(A),person(B),
             h(f(B,A)).
```

3. Murderers are not innocent.

```
:- h(i(P)),m(P), person(P).
```

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

```
:- person(P), person(P1),
   h(o(P)),h(t(P,P1)).
```

5. Friends know each other.

```
:- person(A), person(B),
   h(f(A,B)),h(nk(A,B)).
```

6. Person who was out of town cannot be a murderer.

```
:- h(o(P)), m(P), person(P).
```

7. Exactly one of the suspects is a murderer.

```
1{m(a),m(b),m(c)}1.
```

The last two statements are SMODELS directive which tell SMODELS to suppress the display atoms different from those formed by predicate m**.**

hide. show m(F).

The only answer set of the program contains $m(b)$ **correctly concluding that Bertram is the murderer.**