Course on Artificial Intelligence and Intelligent Systems

Exercises on Abductive Reasoning in Prolog and CHR

Henning Christiansen
Roskilde University, Computer Science Dept.

1 Exercise: Planning for Construction Works

The purpose of this exercise is to show howabduction can be applied for planning of complex projects. From some initial state (defining available resources, etc.), the goal is to reach a successful final end of project, and the plan is, then, a specification of actions and their order that will lead to that goal. Let us consider the project of building a little two-dimensional Greek temple of the following form.

```
                     ====
                  ==========
              =====gable======
           ======================
        =============================
         ||||                  ||||
         ||||                  ||||
         ||||                  ||||
         ||||                  ||||
         |c1|                  |c2|
         ||||                  ||||
         ||||                  ||||
         ||||                  ||||
         ||||                  ||||
      ---------------------------------
      |               f1               |
   ---------------------------------------
   |                  f0                  |
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 ~~~~~~~~~~~~~~~~~~~~soil~~~~~~~~~~~~~~~~~~~
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

It is constructed from five parts, identified as `[gable,c1,c2,f1,f2]`, and when building the temple, inherent physical constraints must be taken into account. For example, column `c2` is supported by the top layer `f1` of the foundation, so a plan that suggests to put `c2` in its place before `f1` should be rejected. The `soil` is already there and we can assume for homogeneity that it was placed in the universe at time `0`.

One way of viewing this planning task is that we need to put each part it its place (i.e., we are finished when all parts are in place), provided that inherent constraints for the plan are satisfied. We can assume that the first part is placed at time `1`, the next one at time `2`, and so on.

Having solved this exercise, you should have produced a running planning system using CHR+Prolog using the methods introduced in the course. The separate questions below are intended to guide you gradually to a solution.

A little auxiliary predicate

As you are not expected to have a long experience in Prolog programming, you are offered the following predicate which can be applied to take out the elements of a list in an arbitrary order (so that you can get along with the exercise).

Assume that the planning system's knowledge base includes the following fact, which gives the list of parts for our temple (their order is not interesting).

```
parts([gable,c1,c2,f0,f1]).
```

For the construction of a plan we need, at different times, to be able to take out some arbitrary part from this list, and send on the remaining list of components to further processing. For this purpose, we introduce a predicate with the following arguments,

`takePart(`*WhichPart*`,` *Parts*`,` *RemainingParts*`).`

So if we call, say, `parts(P,[gable,c2,f1],Rest)`, we may get the different solutions under backtracking (three, in fact, for this example), one which is `P=c2, Rest=[gable,f1]`. ((You may notice that the standard `member` predicate also can be used to generate different members, but it lacks the ability to return the list of remaining elements.))

Here we show a definition of `takePart` which is based on `append` which in SICStus Prolog needs to be imported as part of a library.

```
:- use_module(library(lists)).

takePart(X,List0,List1):-
    append(LeftRest,[X|RightRest],List0),
    append(LeftRest, RightRest,List1).
```

1.1 Database describing the architect's design

Construct a little database of Prolog facts that describes the aspects of the picture above that are necessary for solving the planning task. (The `parts` predicate shown above can also be considered part of this database.)

Type it into the computer and test it with some simple queries.

1.2 Abducibles and their integrity constraint(s)

Abduction means to specify the goal or observation, and the task is to figure out that basic facts ("abducibles") that can explain this goal. For planning tasks, the abducibles are individual steps of the plan, with an indication of at which relative time they should take place (see about time above).

Define using CHR these abducibles as constraints, and write one or more integrity constraints as CHR rules, that you believe are necessary for accepting sensible plans and throwing away the bad ones. As the plan is likely to be built up step by step, it is practical that these constraints works in a correct way also for partial plans. (Hint: It is possible to solve with a single CHR rule, which needs a guard.)

Type the solution into the computer and test it with some simple queries formed by abducibles (that you provide yourself).

1.3 The "driver algorithm"

The final component that you need to provide for the temple building planning system, is to define a predicate that describes what is meant by a plan, and how the initial state for the planning is defined and when a plan is finished. We have indicated all information in the introduction to this exercise, and your job is now to write it into Prolog.

Hint: One way to give a solution is to define a Prolog predicate `build(`*Parts*`, `*Time*`)` which makes one step of the plan and then generate a recursive call to solve the remaining problem.

Final remark

This exercise showed abduction applied for planning problems. You should be aware that methods for planning problems has been studied for several decades, and there exist multitudes of systems and algorithms for such purposes.

The method indicated in this exercise has very bad scaling properties unlezz additional optimization techniques are introduced.

2 Exercise: Diagnosis of power supply networks

The purpose of this exercise is to apply the abduction-based diagnosis method described in the course to find faults in power supply networks.

We consider a geographical region which has a power supply control centre that gets informed as soon as any village in the region is without electricity power. In such cases, the centre checks, using mobil phones, whether the remaining villages have electricity power. On the basis of this information, the centre must take the decision of where to send the repair teams. Our job is to develop a diagnosis system that given the information of which villages have or have not electricity power, can give suggestions for which wires or power plants that might be down, and thus are candidates for repair.

For this application, it seems reasonable to take the consistent fault assumption, as if a wire is down, it will stay down until it is repaired.

A given wire w will be in one of two states, `state(`w`,up)` or `state(`w`,down)`, so a diagnosis is a combination of abducibles concerning the different wires in the region. There is also a risk that the power plants in the region can go down, so a diagnosis may also contain abducibles such as `state(`*pp*`,down)` where *pp* refers to a power plant.

Observations are sets of "goals" of the form `has_power(`v`)` or `has_no_power(`v`)` where v refers to a village. In a given situation, we may have such information about all or just some villages.

The electricity power network can be depicted as follows; `pp1` is the only power plant in this region, `w1`–`9` are the wires, `n1`–`4` are nodes in which a number of wires are connected, and `v1`–`v4` are the villages.

```
  ~~~~~~

 ||                                                     --
#######    w1           w2           w3           w4   /||\
##pp1##---------(n1)---------(n2)---------(n3)---------[v3]
#######           |            |            |
#######           | w5         | w6         | w7
                  |            |            |
                  |           --           --
                 (n4)        /||\         /||\
                /   \        [v1]         [v2]
            w8 /     \ w9
              /       \

             --       --
            /||\     /||\
            [v4]     [v5]
```

For simplicity we assume that nodes always are working (so no `up` or `down` information for those). However, it is convenient to define the `has_power` and `has_no_power` predicates so they take as arguments both villages and nodes.

These predicates should, then, be defined as Prolog predicates (for the set of nodes and villages) that depend on the `state` of incoming wire and the possible `has_power` or `has_no_power` for the previous node.

2.1

Implement the indicated diagnosis system by defining:

- constraints corresponding to abducibles together with the necessary integrity constraints,
- predicate definitions for the `has_power` and `has_no_power` predicates.[1]

To make testing easier, it may be suggested that you define test predicates such as the following.

```
observe_all_butV3:-
  has_power(v1), has_power(v2), has_no_power(v3),
  has_power(v4), has_power(v5).

observe_all_butV1V3:-
  has_no_power(v1), has_power(v2), has_no_power(v3),
```

[1]Notice that you need to define both `has_power` and `has_no_power` separately since negation cannot be used, i.e., if you try to define `has_no_power` by the negation of `has_no_power`, it won't work.

```
  has_power(v4), has_power(v5).

observe_V4V5:-
  has_no_power(v1), has_no_power(v2), has_no_power(v3),
  has_power(v4), has_power(v5).

observe_total_darkness:-
  has_no_power(v1), has_no_power(v2), has_no_power(v3),
  has_no_power(v4), has_no_power(v5).
```

Implement and test you systems.

2.2

If you have succeeded in getting your solution to the previous question to work, you may try to make the task more complicated by adding an extra power plant `pp2` which is connected by a new wire `w10` to the node `n4`. The following assumptions are made.

- `pp2` is for emergency situations only in which power plant `pp1` goes down. We assume some fault-free (!) communication equipment that makes sure that `pp2` cannot be up if `pp1` is up, and the other way round. (But be prepared for the worst case when both power plants go down).
- Wire `w5` gets the special function, that if `pp1` is up, electricity runs from `n2` to `n4`; if `pp2` is up, it is the other way round.

Take the last assumption carefully into account when you revise the definitions of `has_power` and `has_no_power`; otherwise you may easily program infinite loops.

3 Programming project

Extend the temple building in a the following ways (one thing at a time):

1. Each step in the plan takes a certain amount of time to execute. For example, it may take different times to raise each of the two columns; the raising of both can take place at overlapping time intervals after that foundations are finished; however, the gable cannot be added before that last of the columns have been raised. Extend the program so that it take this aspects into account; you should add to the database component of the program, information about the time expected for each possible step.
2. Each step in the plan requires a number of workers to be performed. However, we may assume that we only have a fixed number of workers available, so this limits how many steps can be active at any given moment.

It may be the case that you need to extend the application to a more complex building in order to test your new program(s) in interesting ways.