SOW-BKI329 Representation and Interaction (2017-2018)
Practical Assignment 1

J. Kwisthout and P.C. Groot

1 Assignment 1: Consistency-based diagnosis

Model-based reasoning is one of the central topics of knowledge representation and reasoning in artificial intelligence. The second part of this assignment aims at testing your understanding of Prolog and model-based reasoning by implementing the hitting-set algorithm that you have seen during the lecture.

1.1 Working with trees in Prolog

In order to implement the hitting-set algorithm, we need to be able to represent trees in Prolog. The first task helps to get used to working with such data structures in a logic programming environment.

Exercises In contrast to imperative programming languages, Prolog does not contain constructs for "real" data structures in the language. However, using *terms*, it is possible to represent any data structure by a compound term that constructs such a data structure. While you have seen some basic data structures before (e.g. lists), the following exercise illustrates this further using tree structures in Prolog. You do not need to submit these warm-up exercises and you may ask the instructors for help, if necessary.

(a) Consider for example a binary tree which we can build up using the following two terms (some examples are given below):

- a constant `leaf` which represents a leaf node;
- a function `node` with arity 2, which, given 2 nodes (its children), returns a tree.

In logic programming, there is no need to define these terms somewhere in your programme other than using them in your functions. Keep this in mind when you start programming. Now try and define a predicate `isBinaryTree(Term)`, which is true if and only if `Term` represents a tree. For example, the query `isBinaryTree(Term)` should return true if `Term = leaf`. Test this by using the query isBinaryTree(Term) and using the following compound terms such as:

- `leaf` (true)
- `node(leaf)` (false)
- `node(leaf,leaf)` (true)
- `node(leaf,node(leaf,leaf))` (true)

(b) Define a predicate `nnodes(Tree, N)`, which computes the number of nodes N of a given `Tree`, e.g.

```
?- nnodes(leaf,N).
N = 1.

?- nnodes(node(leaf,node(leaf,leaf)),N).
N = 5.
```

(c) Extend the representation of the tree so that each node (and leaf) is labelled with a number. This number can be given as an argument to a tree component (node/leaf). Adapt your definition of `isBinaryTree` and `nnodes` to reflect this representation. Note: Expanding the arguments that leaf and node hold is the main purpose of this exercise. These arguments are not filled by Prolog by calling upon this function, but can be filled by you to check i.e. whether your extension of `isBinaryTree` works (see below for an example query).

```
?- isBinaryTree(node(2, leaf(1), node(1, (leaf(0), leaf(0)))).
```

(d) Define a predicate `makeBinary(N, Tree)` which gets some number $N \geq 0$ and returns a tree where the root node is labelled by N. Furthermore, if a node is labelled by $K > 0$, then it has children that are labelled by $K - 1$. If a node is labelled by 0, then it does not have children and is a leaf. In other words: all nodes on the same depth-level should have the same label and the depth of the whole tree is equal to the root node label, N.

(e) Now define a predicate `makeTree(N, NumberOfChildren, Tree)`, which gets some number $N \geq 0$ and some number $NumberOfChildren \geq 0$ and returns a tree. The depth of the returned tree should be equal to N and each node in the tree should have $NumberOfChildren$ children. We are dealing with symmetric Trees, since the $NumberOfChildren$ value stays the same throughout the function. Hint: the `makeBinary` representation won't suffice anymore, but it can be used to draw inspiration from. However, the children of nodes will have to be specified in a different way, namely lists. To build your Tree and systematically add children in a recursive manner, creating a help function to do this might be very useful.

1.2 Implementation of the hitting-set algorithm

The *hitting-set algorithm* acts as the core of consistency-based diagnosis, and has been discussed during the course. In these tasks, you will implement this algorithm in Prolog.

Task 1: Generate conflict sets To get started, perform the following exercise.

- Download `tp.pl` and `diagnosis.pl` from blackboard. Load them into Prolog.
- In `tp.pl`, scroll down to the bottom and inspect the definition of `tp/5`. The `/5` stands for the number of arguments the function `tp` takes.
- In `diagnosis.pl`, inspect the definitions of the diagnostic problems in the file. Formulas are represented by Prolog terms where constants (and functions) are interpreted as predicates, with additional operators `~` (*not*), `,` (*and*), `;` (*or*), `=>` (*implies*), `<=>` (*iff*), and quantification $\{$`all`, `or`$\}$ `X:`f, where `X` is a (Prolog) variable and f is a term which contains `X`. Since `,` and `;` are also Prolog operators, it is often required to put brackets around these terms. For example, the formula $\forall_x(P(x) \vee Q(x))$ can be represented by the term `(all X:(p(X) ; q(X)))`.

Experiment with `tp` and determine at least three conflict sets for the diagnostic problems. The query written below obtains and uses the System Description (SD), Components (COMP) and Observations (OBS) to generate conflict sets with `tp` for `problem1`.

```
?- problem1(SD, COMP, OBS), tp(SD, COMP, OBS, [], CS).
```

For one out of the three problems, provide proof that one of its conflict sets is indeed a conflict set. An informal proof with words will suffice, but you should show your understanding of what a conflict set is.

In your report, describe the input and output of `tp`, what `tp` is used for in your project and when.

Task 2: Implementation Implement the hitting set algorithm. The algorithm can be split into three building blocks. The three predicates described below will help you to implement the algorithm in a structured way:

- Define a predicate `makeHittingTree`, that generates a hitting tree for a diagnostic problem. Structure is key, so think well about how you want to build up your tree and take the images from the powerpoint slides as your guideline. Just like in 1.1e) it might be useful to define a predicate that helps you adding the children of a node to the tree.
- Now that your program is able to generate hitting trees you should extend your code such that the diagnoses for a diagnostic problem are read from the hitting tree leaves. For this you can define the predicate `gatherDiagnoses`. Use a list to store and update your results.
- Through `gatherDiagnoses` your code should have stored all diagnoses as a list of lists. However the aim of the hitting set algorithm is to find only the minimal diagnoses for some diagnostic problem. Define the predicate `getMinimalDiagnoses` that returns a list of minimal diagnoses. This means that from the original diagnosis-list all supersets should be removed. Tip: Removing supersets from a list of lists is a common task (even in Prolog), clear examples and functions can be found on the internet.

In the end, the use of these three predicates should be combined together in one predicate (think of it as a sort of "main" function). This predicate should get a diagnostic problem as an input (thus again a System Description, the Components of the system and some Observations) and return the minimal diagnoses for this diagnostic problem.
In your report you should evaluate your program by using the given diagnostic problems and determine the minimal diagnoses. Explain why the obtained results are correct (or not correct). Also, reflect on your code (such as: What are the limitations? How can it be improved? What are the problems encountered? Do the (optional) optimizations help? What can you say about its complexity?)

(not required) Testing the performance Optionally, add some of the optimizations to prune the search space as described in [1] (see Google scholar or the library for the paper). Modify the example problems to become larger and more complex and investigate the time and space limits of your hitting-set implementation and Prolog (again up to a 10% bonus on your grade; note that the final grade cannot exceed 100).

References

[1] R. Reiter (1987). A theory of diagnosis from first principles. *Artificial Intelligence*, **32**, 57–95.