SOW-BKI329 Artificial Intelligence: Representation and Interaction Practical Assignment 1

P.C. Groot and P. Lanillos

Assignment: Consistency-based diagnosis

Model-based reasoning is one of the central topics of knowledge representation and reasoning in artificial intelligence. 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.

This document is structured as follows. Section 1 provides some pointers on getting help from the Prolog system and compares Prolog with other languages. Sections 2 and 3 provides two alternatives for completing the assignment (i.e., choose one). Both sections contain a number of warm-up exercises for getting started, which do not need to be handed in.

1 Getting Started with Prolog

1.1 Documentation and Notation

Run `help(nameeofpred)` in the SWI-Prolog REPL to get the description of a predicate:

```
?- help(length).

length(?List, ?Int)                                              [ISO]
    True if Int represents the number of elements in List.   This
    predicate is a true relation and can be used to find the length
    of a list or produce a list (holding variables) of length Int.
    The predicate is non-deterministic, producing lists of increasing
    length if List is a partial list and Int is unbound.   It raises
    errors if

      o Int is bound to a non-integer.

      o Int is a negative integer.

      o List is neither a list nor a partial list.   This error
        condition includes cyclic lists.

    This predicate fails if the tail of List is equivalent to Int
    (e.g., length(L,L)).
```

The question marks before the parameters indicate[1] that they can both be used as input as output. Many predicates have fixed input and output, making them more similar to functions in other programming

[1] This is just documentation notation. Including question marks, pluses or minuses in your predicate definitions does not have the intended effect.

languages:

```
?- help(subtract).

subtract(+Set, +Delete, -Result)                                    [det]
    Delete all elements in Delete from Set. Deletion is based on
    unification using memberchk/2. The complexity is |Delete|*|Set|.
    A set is defined to be an unordered list without duplicates.
    Elements are considered duplicates if they can be unified.

         See also ord_subtract/3.
```

If you don't know the exact name of a predicate use `apropos(Pattern)` to display all predicates and functions that have *Pattern* in their name or summary description.

1.2 Analogies to Prolog

If a predicate does not require any branching, pattern-matching, or backtracking, its structure should be similar to an imperative function. Take this Python function:

```
def pythonfunc(input1, input2):
   res1 = builtin(input1, input2)
   res2 = helper(res1)
   output = finalize(res2)
   return output
```

This would be the Prolog equivalent:

```
prologfunc(Input1, Input2, Output) :-
   builtin(Input1, Input2, Res1),
   helper(Res1, Res2),
   finalize(Res2, Output).
```

Note that the return value is included both in the predicate header and the final expression. Recursing over a list is very similar to doing the same in Haskell or Clean:

```
max_helper([], Highest, Highest).
max_helper([X | Xs], CurrentBest, Highest):-
   X > CurrentBest,
   max_helper(Xs, X, Highest).
max_helper([X | Xs], CurrentBest, Highest):-
   X =< CurrentBest,
   max_helper(Xs, CurrentBest, Highest).

max_of_list([X | Xs], Output):-
   max_helper(Xs, X, Output).
```

2 The Hitting Set Algorithm Using Tree Structures

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.

(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.

2.1 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 Brightspace. 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 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 code you should also make use of the `tp` function you used in the previous task.

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?)

Task 3: (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).

Task 4: Complete your report Write a report for your assignment. Evaluation criteria are provided in a separate document.

3 The Hitting Set Algorithm Without Explicit Tree Structures

The Hitting Set Algorithm can also be implemented without explicitly using a tree structure but by using recursion and implicit branching on choice points. It is hypothesized that implementing the hitting set algorithm without explicit tree structures will both be easier for most students and result in a greater understanding of Prolog. However, this hypothesis is untested. Unlike this section, the explicit route above has been around for many years, and while it is considered very difficult by students, the vast majority ends up with a working program.

If you wish to take the implicit route, it is recommended that you first complete some or all of the exercises below, which serve as an introduction to branching and recursion in Prolog. Finding hitting sets should be much easier once you have completed the path finding exercise. The collection exercise should prepare you for obtaining the minimal hitting sets.

Exercises You do not need to submit these warm-up exercises and you may ask the instructors for help, if necessary.

(a) Reimplement the `member/2` predicate. `member(?Elem, ?List)` returns true if `List` contains `Elem`. It can also be used to branch on all elements in `List`:

```
?- member(a, [a,b,c,d,e]).
true ;
false.

?- member(f, [a,b,c,d,e]).
false.

?- member(X, [a,b,c,d,e]).
X = a ;
X = b ;
X = c ;
X = d ;
X = e.
```

It is not necessary to keep both use cases in mind while implementing this behavior. If your predicate can do one of these things, it can probably also do the other.

(b) Reimplement `subset/2`. `subset(+Sub, +Set)` succeeds if all elements in `Sub` are in `Set`.

(c) Implement `path(+From, +To, +Graph, +Visited, -Path)`, which returns all paths from `From` to `To` in the undirected graph `Graph`:

```
?- path(6, 1,
[edge(1, 5), edge(1, 7),
```

```
edge(2, 1), edge(2, 7),
edge(3, 1), edge(3, 6),
edge(4, 3), edge(4, 5),
edge(5, 8),
edge(6, 4), edge(6, 5),
edge(7, 5),
edge(8, 6), edge(8, 7)],
[], Path).
Path = [6, 4, 3, 1] ;
Path = [6, 4, 5, 8, 7, 1] ;
Path = [6, 4, 5, 8, 7, 2, 1] ;
Path = [6, 4, 5, 1] ;
Path = [6, 4, 5, 7, 1] ;
Path = [6, 4, 5, 7, 2, 1] ;
Path = [6, 5, 8, 7, 1] ;
Path = [6, 5, 8, 7, 2, 1] ;
Path = [6, 5, 1] ;
...
Path = [6, 8, 5, 7, 2, 1] ;
false.
```

The graph is defined as a list of edges. You do not have to define the `edge/2` predicate. Because Prolog is homoiconic, you can use just about any valid syntax as a data structure:

```
?- member(my_edge(From, b), [my_edge(a, c), my_edge(a, b)]).
From = a.

?- member(a + b, [a + b, a + c]).
true ;
false.

?- member(b + a, [a + b, a + c]).
false.
```

Using an addition sign is not especially recommended, of course. To implement `path/5` incrementally, first write a predicate which only succeeds if a path between From and To exists, without worrying about output. It may also be simpler to initially treat the input as a directed graph without loops.

(d) Usually, predicates are unable to access multiple results from another predicate at once. In previous exercises, we could rely on implicit backtracking to take care of the forking paths because it did not matter if we "forgot" the other solutions in the meantime. This is not always the case. Fortunately, `findall(+Template, +Predicate, -List)` allows us to obtain all solutions for a single predicate. `findall/3` is a meta-predicate, which is comparable to a higher order function in functional programming. Consider this implementation of set intersection:

```
in_both(Elem, A, B) :-
member(Elem, A),
member(Elem, B).

intersection(A, B, Intersection) :-
findall(Elem, in_both(Elem, A, B), Intersection).
```

For each solution of `in_both(Elem, A, B)`, the resulting Elem is added to Intersection.

```
?- intersection([1,4,7], [3,5,4,7], Res).
Res = [4, 7].
```

Create a predicate `longest_paths(+From, +To, +Graph, -Paths)` which outputs a list of the longest paths from `From` to `To`. This predicate should call `path/5`. Run `help(length)` and `help(>)` for additional documentation.

3.1 Implementation of the hitting-set algorithm

Task 1: Generate conflict sets This is identical to Task 1 in Section 2.1.

Task 2: Implementation Implement the hitting set algorithm using recursion and implicit branching on choice points. In the end, everything 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 code you should also make use of the `tp` function you used in the previous task.

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?)

Task 3: (not required) Testing the performance This is identical to Task 3 in Section 2.1.

Task 4: Complete your report Write a report for your assignment. Evaluation criteria are provided in a separate document.

Acknowledgement

The assignment has been around for a couple of years (Section 2). Thanks to Bob de Ruiter for extending the document with additional information on getting started with Prolog (Section 1) and providing an alternative solution (Section 3) to the assignment.

References

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