Knowledge Representation and Reasoning
BKI312 (2015-2016)
Practical Assignments (Series 1)

M. van der Heijden and J. Kwisthout

October 2015

1 Introduction to the assignments

Welcome to the first series of assignments on knowledge representation and reasoning. In this series you will program the hitting-set algorithm and you will model and solve planning problems in Prolog[1].

Additional files The files that you need are available from Blackboard (see assignments)

Marks and Time This assignment is marked out of a total of 100 percent, and it contributes a total of 20 percent towards your overall grade for the course knowledge representation and reasoning. The distribution over the sub-assignments is as follows:

Assignment 1-1: 50 percent
Assignment 1-2: 50 percent

The estimated total amount of time for the average student can be roughly four full days of work per student. This varies since most students work in teams of two. The four days are effort asked for, but in case much more is needed to complete only the required parts, contact the teachers. During the planned practical sessions you can work on the problems in the vicinity of the teachers. They can help out with many practical issues, explain Prolog concepts and so on, but remember that this assignment is part of the evaluation (and contributes to your grade) and this means that the teachers will not help you with the essential problem solving parts.

Submission See the individual descriptions of the assignments for detailed descriptions of what to hand in. We expect all written answers in the form of a **report**, including formalizations, pictures, and small code fragments to illustrate your answers. Try to make it readable, so explain what you are doing and why. Each assignment needs to be covered in a separate section (or chapter) of your report.

The report is important. Code is required to answer most of the questions, but the accompanying *descriptions* or *explanations* are equally important for your grade. You need to hand in the code as well as the report. *In addition*: for each of the assignments, answer the following questions: i) how much time did it take you to finish it?, ii) if you would have to change aspects of the assignment: what would they be and why? Code files are to be submitted in a zip-file, properly named. All submissions should be done through Blackboard.

Good Luck!

The deadline for submission is

17th November 2015

[1]Thanks go to M. Crosby for his help on the sitcalc assignment.

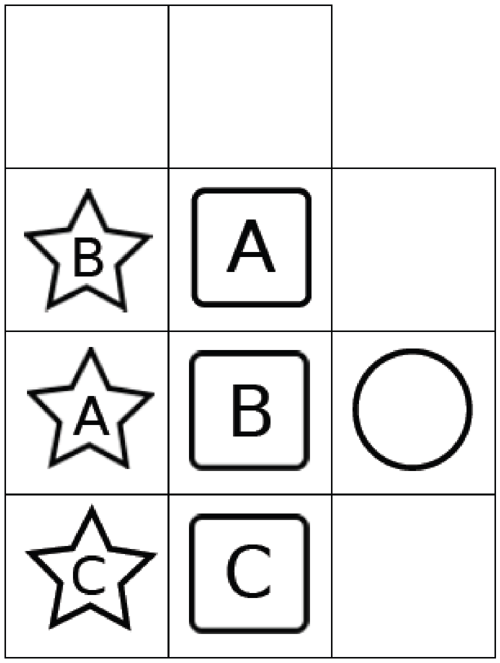

Figure 1: Example Sokoban problem. The grid is a 3x4 rectangle with the upper-right square missing. The agent is represented by the circle. There are three crates (represented by the boxes) A, B and C. The stars represent the goal locations for the respective crates.

2 Assignment 1-1: Situation calculus and planning

This assignment deals with modelling and programming using situation calculus.

2.1 Introduction

This assignment is about the Situation Calculus and planning. It will evaluate your skills in formalizing, implementing and testing a planning problem. Part 1 is a written exercise that requires you to formalize a planning problem using Situation Calculus. Part 2 requires you to implement the model and verify its correctness using a planner. In Part 3 you can extend the model and its implementation to deal with additional aspects of the environment. In Part 4 you need to answer some broader questions.

2.2 Part 1: Modelling Sokoban

The first part of the assignment requires you to develop a model using the Situation Calculus (sitcalc from now on). You will then use the axioms you defined to infer a plan for a simple instance of the problem.

Problem description In the Sokoban domain, an agent moves around a grid world pushing boxes into desired locations. The agent can move to any orthogonally connected empty grid square. Additionally, there are crates in the domain that an agent can push. An agent can push a crate only in a straight line and only if the space behind it is empty. There may only be one crate (or agent) in any grid location at any given time. Figure 1 shows an example problem. It contains eleven grid squares, three crates and one agent. Each crate has a respective goal location that it must be pushed to. We will refer to locations numbered so that the bottom left location is `loc1-1` and the two top-right locations are `loc2-4` and `loc3-3`.

Task 1: Knowledge base The first step in the creation of a model is the design of the knowledge base, i.e. the structures that will hold information about the environment that the planner can use when it chooses an action. The initial model should include information about the grid world, the crates and the location of the player. You should define a set of predicates that can encode every state of the problem. Some of them will be atemporal predicates, which do not change as time progresses, and some will be fluent predicates whose values depend on the current situation. Briefly comment on all predicates you introduce.

Q1: Specify how you would show which locations are connected. This should include the direction in which the locations are connected as this will help when defining the push action.

Q2: Explain how to keep track of the position at which the agent and crates are located at any particular moment and which locations are empty.

Q3: Using the symbols you just defined, write down the initial state of the problem depicted in Figure 1.

Q4: Using the symbols you just defined, describe how to specify the set of goal states of the problem depicted in Figure 1.

Task 2: Actions The agent can move from the space it occupies to any adjacent empty space. Alternatively, it can push a crate in a straight line as long as the space behind the crate is empty. For example, in Figure 1 the only push action the agent can perform (in the initial state) is to push crate B left into `loc1-3` which would leave the crate in `loc1-3` and the agent in `loc2-3`. Formalize the following actions in terms of possibility axioms and effect axioms. You can omit universal quantifiers.

Q5: The agent can *move* to an adjacent empty space.

Q6: The agent can *push a crate* that it is next to into an empty space behind the crate. Note that a crate can only be pushed in a straight line and only into the square directly behind it. This moves the agent into the space the crate originally occupied. You should not need to use any arithmetic operations here. You can make use of the direction information you encoded with your connected predicate.

Effect axioms alone are not sufficient: they describe how the new situation has been affected by the action executed, but they do not update information unrelated to the specific action, which may (or may not, if not updated) remain the same.

Q7: Write the successor-state axioms for the fluents in your model. Briefly describe how you got to these, possibly by highlighting how they could be derived from effect specifications.

2.3 Part 2: Implementation

The second part of the assignment is centred on the implementation of the model you developed in Part 1. Once we have translated the axioms into rules that a planner can understand, we can work on more complex instances of the problem. The conversion is fairly straightforward, mostly a translation of logical symbols into ASCII characters, as we shall see in this section.

A planner and the Situation Calculus You can use the planner in `planner.pl`. We also provide two examples implementing a simple blocks world, `sample-blocks.pl` and `sample-blocks-domain.pl`.

To show the differences and similarities between situation calculus and the language read by the planner, we compare two (simplified) versions of the blocks world example. What follows are the possibility and successor-state axioms for the move action within the blocks world. Following the general conventions, we have predicates starting with an upper-case letter and variables in lower-case, quantified.

$$\forall x, y, s. Clear(x, s) \land Clear(y, s) \rightarrow Poss(Move(x, y), s)$$

In sitcalc, the opposite is true; predicates begin with lower-case letters and variables with capitals. Quantifiers are dropped. Logical connectives change: the implication symbol is now `:-`. A comma represents a conjunction, while disjunctions are marked by semi-colons. The end of a rule is marked by a dot. The following statement means 'if What is clear and Where is clear in state S, then it is possible to move from What to Where in state S':

```
   poss(move(What, Where), S) :-
clear(What, S), clear(Where, S).
```

In logic, a successor-state axiom is guarded by the predicate that verifies if the action is possible.

$$\begin{gathered} \mathit{Poss}(a, s) \to \mathit{On}(x, y, \mathit{Result}(a, s)) \\ \leftrightarrow \\ \mathit{Move}(x, y) \lor (\mathit{On}(x, y, s) \land a \neq \mathit{Move}(y, z)) \end{gathered}$$

This is done automatically by the planner or sitcalc interpreter, and can be dropped. Moreover, we are interested in the planning task, so we keep only one direction of the iff in the formula above: the $\leftarrow$. The resulting sitcalc axiom is (where the word `results` is used instead of `do` in this version):

$$\begin{gathered} \mathtt{on(Block, Support, results(A, S))} :- \mathtt{A} = \mathtt{move(Block, Support)}; \\ \mathtt{on(Block, Support, S), not(A = move(Block, \_))}. \end{gathered}$$

where the semicolon ; is a disjunction, and the underscore is an anonymous variable that unifies with anything.

Task 3: Translate Axioms After reading the sample files and the included documentation, make a copy of the `domain-template.pl` file and rename it `domain-task1.pl`. Translate the axioms of your model and save them in this file.

Simple Experiments The goal of the following three exercises is to learn the language accepted by the planner, and for you to test the correctness of the model. Each task has at least one solution, and all plans do not exceed 15 actions in length; if the planner fails to find a plan, there might be something incorrect in the model. For each task, make a new copy of the file `instance-template.pl`, and rename it `instance-task#.pl`, where # is the number of the task. Any comment or description can go inside the `.pl` source file. Make sure to include both `instance-task#.pl` and `domain-task#.pl` for each task.

Additional hints: there are several simple ways to make planning more efficient. For example, one can add a `visited` fluent to ensure the planner will not look at actions that will lead to an already visited tile. One can also check for *reverse* actions: by checking whether the planner is trying to exactly reverse the previous action in the plan (and preventing that) one can discard unnecessary moves and speed up planning. There are several other tricks one could apply (although they are not necessary to solve the problem per se). If you employ such programming tricks, be sure to describe them properly in your report, and also motivate that they will actually be beneficial for your specific application (either by arguing or by experimental evaluation showing a speedup).

Task 4: The Planning Problem in Figure 1 Implement and test the problem shown in Figure 1.

Task 5: Crates go to Any Goal Location Rewrite the problem so that the crates are allowed to end in any of the goal locations in Figure 1. Implement and test on the same problem.

Task 6: Inverse Problem Find an initial state and goal specification for which the agent visits every location in the grid world in the resulting plan. The agent does not need to revisit its starting location but must visit the goal locations of crates and the crates' initial locations. You may change the number of crates, their locations and their goal locations as well as the agent's starting location (but not the size or shape of the grid world). The goal specification should only include the goal locations of crates (as before). Test whether the resulting plan satisfies the requirements.

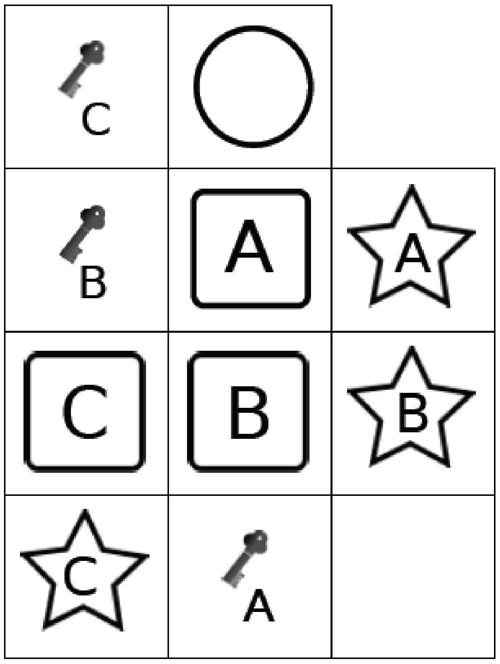

Figure 2: Problem for task 6. A key's label corresponds to the crate it unlocks.

2.4 Part 3: Extending the domain

In this section you will extend the original problem to include new actions, action effects and goals. Only the sitcalc implementation of the axioms is required, but make sure the code is properly commented when defining new predicates. For each task, make a new copy of the file `domain-template.pl`, and rename it `domain-task#.pl`. Any comment or description should go inside the `.pl` source file. Make sure to include both `instance-task#.pl` and `domain-task#.pl` for each task.

Each task is a *separate* extension to the basic problem (Part 2: Tasks 1 and 2).

Task 7: Unlocking the Crates In this version of the domain each crate starts locked to the ground. The agent cannot push the boxes until it has found and picked up the key for each crate. Add a pickup action that lets the agent pick up a key that is in its current location. You should not add any more actions; instead, update your definitions for the push action so that they require the agent to have picked up the correct key. Once implemented, test on the problem shown in Figure 2.

(not required) Various Extensions There are several immediate ways to make things even more interesting. These are not required, but highly encouraged. Completing any of them will be beneficial for your final grade for the assignments, to various degrees (with a maximum of 15 points).

- Extend the model with new exciting effects or new actions. Some blocks may glide multiple grid positions unexpectedly, or sometimes we can jump over a block to get to another position more easily, or ....
- Extend your model with Golog-constructs (look this up: Golog is syntactic sugar to provide more programming-like functionalities in sitcalc). For example a solution can be programmed using `while` *there is an object still not on its place* `do` *something.* More elegantly, one could add procedures to, for

example, *make room* for something else: in order to push some block to its position, one may need to first clear an exit which might take up some actions first.

- Model a larger level of the original game (Google for Sokoban). Originally, the game involved several rooms and corridors. You would need to extend the model to incorporate maybe some additional elements, and surely one needs to be careful with planning since solutions for large levels might become very large. (Extending the model with some Golog-predicates to guide the search for a plan might help too to make planning more efficient).
- Implement a more efficient planner.
- ...(and many other possibilities)

2.5 Part 4: General questions

Task 10: Sitcalc expressivity Reflect on the advantages and disadvantages of sitcalc, using your sitcalc model for the Sokoban domain as an example. Consider things like the elements of the models, the size of models, the relative ease of reasoning (planning) and so on. An additional aspect you should discuss is how easy it is to *extend* the models to new situations (as you did for example to handle keys). A conceptual comparison with e.g. STRIPS (for planning) or other action languages (Event calculus; Temporal Action Logic) could be useful here to put things in perspective.

Task 11: Related work Logical systems for actions and change are increasingly used in modern robotics in AI. Look up a paper (on the web, or in Google Scholar: `http://scholar.google.com`) published after 2010 in which logic or Prolog is used to reason with change, actions or plans in a robotic(-like) setting. Give a correct reference to the paper in your report (i.e. we expect a bibliographic reference, and not just a weblink) and explain briefly the main highlight(s) of the paper. Useful keywords to search include: logic, action, change, Prolog, STRIPS, situation calculus, robotics, relational, roboearth, knowrob, ADL, PDDL, etc. Note that we do not require you to summarize the complete paper for this assignment.

3 Assignment 1-2: 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.

3.1 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.

Define a predicate `isBinaryTree(Term)`, which is true if and only if `Term` represents a tree. For example, `isBinaryTree(Term)` should be true if `Term = leaf`. Test this on 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. Adapt your definition of `isBinaryTree` and `nnodes` to reflect this representation.

(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.

(e) Now extend the representation of your tree so that each node can have an *arbitrary* number of children using lists. Also define a `nnodes` predicate for these kind of trees.

3.2 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 12: Generate conflict sets To get started, perform the following exercise.

- Download `tp.pl` and `diagnosis.pl` from blackboard.
- In `tp.pl`, scroll down to the bottom and inspect the definition of `tp/5`.
- 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/5` and determine at least three conflict sets for the diagnostic problems. For one of these provide a proof that it is indeed a conflict set, and give an informal explanation how `tp/5` computes those sets.

Task 13: Define your data structure Define a Prolog representation for hitting-set trees. Program a corresponding predicate `isHittingSetTree(Tree)` and convince yourself that this predicate is true if and only if `Tree` is a hitting-set tree. Guarantee that the edge labels on a path from the root to another node are distinct. In the following, the `tp/5` program will deliver these labels (conflict sets) for you.

Task 14: Implementation Use this representation to develop a Prolog program of the hitting-set algorithm. The input to the program is a diagnostic problem; the output is the set of all *minimal* hitting sets (i.e. diagnoses). Optionally, add some of the optimizations to prune the search space as described in [1] (see Google scholar or the library for the paper).
You may use standard Prolog functions such as `subset` and `append`. Other useful predicates may include `var(X)` to test whether X is a variable or `=..` to construct and deconstruct a term into symbols. See the SWI-Prolog manual for more details.
Evaluate your program using the given diagnostic problems and determine the minimal diagnoses. Explain why the results are 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 Modify the example problems to become larger and more complex. Investigate the time and space limits of your hitting-set implementation and Prolog.

References

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