Package delivery

– Prolog lab assignment 1 –
– D7012E Declarative programming –

Håkan Jonsson
Luleå University of Technology, Sweden

May 5, 2023

1 Introduction

Consider a small building with three interconnected rooms r_1, r_2, and r_3. To go between r_1 and r_2 (either way) requires a steel key and to go between r_1 and r_3 requires a brass key. It is not possible to move directly between r_2 and r_3. The room r_1 contains the steel key and a mobile two-handed robot, while r_2 contains the brass key and r_3 contains a package (see Figure 1).

The robot can move between rooms, take up items, and drop items. An item is either a key or the package. Dropped items end up in the room where the robot is. To take up an item, the robot must be in the room that currently contains the item. Moreover, the robot is two-handed and can carry at most two items at a time. In contrast, there is no limit on the number of items a room can contain.

The question is now if the robot can fetch the package from room r_3 and bring it to room r_2? If so, how should the robot behave?

2 Assignment

Write a Prolog program that computes answers to the questions above. If it is possible for the robot to deliver the package, your program should produce a list with the actions needed given in the order they should be carried out.

Implement a clause `solveR(State,N,Trace)` that, given a state `State` and a positive integer `N`, tries, going no deeper into the state graph that `N` steps, to find a solution in the

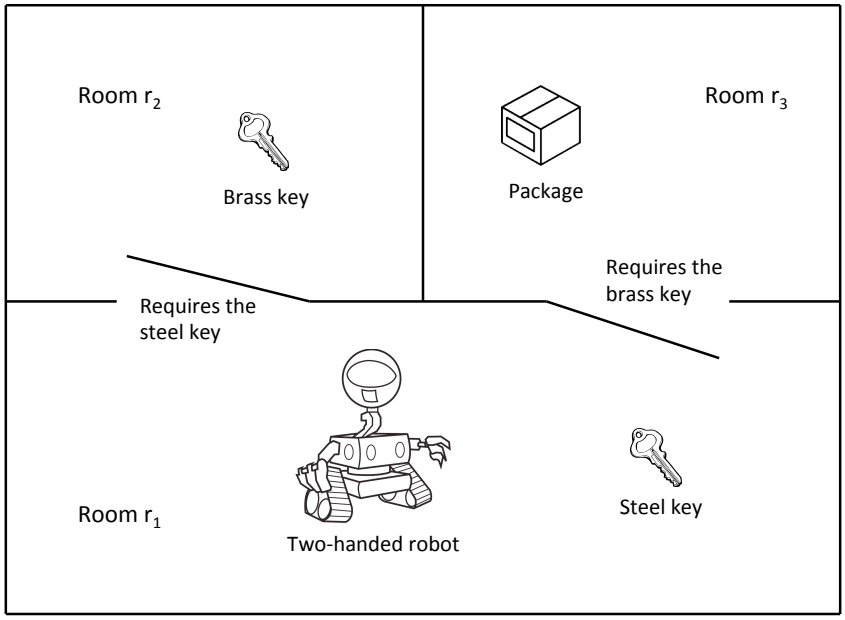

Figure 1: The interior of the building at the start.

form of a list `Trace` of actions taken by the robot. If such a solution can not be found, the goal should fail.

You are free to choose whatever representation of the state as you see fit.

3 Examples

For inspiration, a bundle with a number of Prolog solutions based on the same AI technique (*State Space Search*), but for different problems, have been posted on the course web page. Here are brief descriptions of them. Much more can be found on the internet. There are also comments in the programs containing further explanations.

3.1 monkey.pl

This is a solution to the problem *Monkey and Banana* that has been shown and discussed in class.

`https://en.wikipedia.org/wiki/Monkey_and_banana_problem`

The program has been adjusted so it returns all the actions the monkey performs (as a list).

3.2 farmer1.pl

This solution solves the farmer's problem of crossing a river with a fox, a goose, and a bag of beans:

`https://en.wikipedia.org/wiki/Fox,_goose_and_bag_of_beans_puzzle`

This used to be the third assignment and the old instruction has therefore been included in the bundle (named `farmer.pdf`), if someone likes to read what lead to `farmer1.pl`.

The program is similar to `monkey.pl` in that it collects the pieces of a solution in a list. It is, however, different in that it takes the maximum recursion depth as a parameter. The implementation also contains an explicit procedure (`invalid`) that checks possible state transitions. The part

```
\+(invalid(NewState))
```

is "negation as failure". A goal `\+goal` succeeds when `goal` fails.

3.3 farmer2.pl

Another solution to the farmer's problem that is very similar to `farmer1.pl` but a bit more verbose. It also has a procedure that stops illegal state transitions.

3.4 jugs.pl

This is a solution to problems where we have two jugs[1] and like to know if there is a way to pour liquid in and out of them so that, in the end, they contain certain given amounts. Our problem is a 2-jugs version of the general `Liquid water pouring problem`:

`https://en.wikipedia.org/wiki/Liquid_water_pouring_puzzles`

This program first introduces the 8 state transitions there are. Then follows three general recursive procedures that, when successful, in combination return a list with the state transitions from the initial state to the end state. The procedures perform a `depth-first search` in the implicit state graph and has two interesting properties:

1. It prevents cycling in the graph. This is accomplished by keeping track of all states that have been visited. The result is a more efficient program that can not get stuck going around in circles in the graph.

2. Like in the programs `farmer1.pl` and `farmer2.pl`, there is a maximum recursion depth. However, it is not a parameter of the program but something automatically incremented by the program until a solution has been found. It starts with maximum depth 1 and re-tries with maximum depths 2, 3, 4, and so on until a solution is found. To be able to stop the program, I have added the possibility to activate a hard limit on the recursion depth.

However, a drawback with this program is that it does not explicitly give us the actions performed. These must be figured out by comparing consecutive states in the result.

[1] A jug is a "kanna" or "tillbringare" in Swedish.