Lisp – GPS operators & search

objectives

1. use tuples to specify state descriptions
2. develop simple GPS style operators
3. develop GPS style operators using matching forms
4. investigate limitations simple-search strategies

In this set of tutorials you will develop small virtual world descriptions using sets of tuples and GPS-style operators for transforming states. These operators will be based on the techniques introduced in lectures. The first operators you define will not use pattern-matching, matching will be introduced after you have experimented with non-matching operators. Similarly the virtual world you define will start very small & acquire extra features as you progress

Before starting you should read your lecture notes on GPS style operators (most text books will also describe these operators).

background

A virtual world environment contains 3 rooms with doors between them (a hall connects a kitchen and a bedroom). There is a cupboard and a table in the kitchen and a bed in the bedroom. There is a robot in the world. The robot has some kind of arm which allows it to hold (at most) one item, it also has some device which allows it to climb on some bits of furniture like tables & chairs (use your imagination).

In order to follow this tutorial you need to load the ops-search function available from the web page. You must also load the matcher & utils.

Your first task will deal with a situation where the robot needs oiling but the oil is locked in the cupboard in the kitchen and the cupboard key is on the bed in the bedroom.

As an example consider the problem of the robot collecting the key. In this example the robot starts standing next to the bed with nothing in its hand and finishes when the robot is holding the key, ie:

```
start:  ((at R bed)(has R nil)(on key bed))
goal:   ((has R key))
```

Minimally: 2 operators are required for this (i) climb-on-bed (ii) get-key.

climb-on-bed is possible when the robot, R, is next to the bed (at R bed) and its effects are to move the robot from being at the bed to being on the bed;

get-key is possible when the robot is on the bed, its hand is empty & the key is on the bed, its effects are to remove the key from the bed & put it into the hand of the robot.

The Lisp definitions are as follows...

```
(defparameter *ops*
  '((climb-on-bed
     (txt . (R climbs on bed))
     (pre . ((at R bed)))
     (del . ((at R bed)))
     (add . ((on R bed)))
     )
    (get-key
     (txt . (R gets key))
     (pre . ((has R nil)(on R bed)(on key bed)))
     (del . ((has R nil)(on key bed)))
     (add . ((has R key)))
     )
    ))
```

calling search

Using the ops-search fn is slightly different to using breadth-search earlier tutorials because we don't want to have to exhaustively specify the goal (in fact there are reasons why we *cannot* do this with larger problems – think about it). Instead we give the search mechanism a subset of a virtual world to check for (ie: we give it a small set of tuples which we want to exist in any goal, any tuples we don't specify are those we don't care about). The goal subset for the small problem above is ((has R key)).

ops-search is called as follows, check the output & ensure you understand its structure. The output is a list of states & their partial paths between start & goal. The goal is shown 1st working backwards to the start state...

```
> (ops-search '((at R bed)(has R nil)(on key bed))
              '((has R key)) *ops1*)
(((path (R climbs on bed) (R gets key))
  (has R key) (on R bed))
 ((path (R climbs on bed))
  (on R bed) (on key bed) (has R nil))
 ((at R bed) (has R nil) (on key bed)))
```

task 1

The robot needs oiling but the oil is in the cupboard in the kitchen which is locked. The robot is standing in the bedroom, the key is back on the bed. The goal is for the robot to get the oil.

Define the extra operators necessary to solve this problem & ensure it works by calling breadth search. Assume (for now) that there are no doors between the rooms.

Note: the cupboard door can be in different states (open, closed and locked), your state descriptions need to represent these different states and your operators manipulate them in a consistent way.

task 2

The objective for the following tasks is to define operators which use matching expressions in their definitions (NB: the world will get more complex as you progress through the following tasks).

First we want to produce generalised *climb-on* and *climb-off* operators. Objects like the bed and the table are *climbable*, the cupboard is not. The robot can climb on an object if the object is climbable and the robot is next to it. Extend your state descriptions to contain information about objects climbability – add tuples like (climbable bed) and write operators for *climb-on* and *climb-off*.

task 3

Doors between rooms and on cupboards, etc can be (i) locked (ii) unlocked but closed (iii) open. In order to change the state of a door the robot needs to be next to it and to lock/unlock it the robot needs a key (or some other suitable instrument) – you can represent this with tuples like (locks key3 door2).

Write general purpose operators (ie: not specialised to particular instances of door) to deal with lock, unlock, open & close.

Think carefully about your state descriptions. Separate tuples describing static features of the world (those which will always remain true) from those which may change. Static features are things like (climable chair) and (locks key3 door2), dynamic features are those which the operators can change.

Collect together your static feature tuples and bind them to a single variable, eg:

```
(defparameter *world1*
      '((locks key3 door2) (in table kitchen)
        (climable chair)   (climable table)  ))
```

Use this world description as follows in the call to *ops-search* (it will allow the search to run more efficiently)...

```
(ops-search start goal operator-list :world *world1*)
```

task 4

The robot can move (i) from place to place within a given room and (ii) between rooms. The differences between these types of move operations are significant enough for you to define two separate operators for this.

You will have to extend your static state information to handle the second operator because you will need tuples stating which two rooms any door connects. There are various ways you can represent this information, one possibility is to use a quadtuple like...

```
(connects bedroom-door bedroom hall)
```

If you take this approach you should remember to include both complimentary forms of relations like **connects** in your state descriptions, this will make the operators easier to write, ie: use both...

```
(connects bedroom-door bedroom hall)
```

and

```
(connects bedroom-door hall bedroom)
```

NB: when you have covered the forward chaining inference mechanism we will consider how to use it to produce (r x y) given (r y x).

task 5

Write operator(s) *pick-up* which allows the robot to pick something up if either (i) it is next to it, ie: (at R ?place) and (at ?thing ?place) are both true or (ii) the robot is on the same piece of furniture that the object is on, ie: (on R ?place) and (on ?thing ?place) are both true.

Write operator(s) drop which allows the robot to drop an item it is holding.

task 6

The complete scenario which uses the ideas above is as follows... there are 3 rooms: a kitchen which connects to a hall which in turn connects to a bedroom. There is a cupboard in the kitchen, a bed in the bedroom and a table in the hall. The cupboard key is on the bed and the kitchen door key is in the hall. To pick things off the table and/or the bed, the robot must climb onto the table/bed first. The door between the kitchen & the hall and between the hall & the bedroom are both closed but not locked. The cupboard in the kitchen contains a bottle of robot-oil, the cupboard door is locked. The robot is in the hall and needs to get the oil.

Consider this problem, separate tuples describing static features of the world from dynamic features and ensure that your operators are adequate to solve this problem.

Carefully analyse the use of your operators with the breadth-first search mechanism and estimate the number of nodes/states that the search will generate. Do this on paper rather than through practical experimentation. Consider the effect of adding more rooms, furniture and manipulable items to the problem space. In particular consider what happens to the problem if we put a bunch of grapes on the kitchen table (the robot can pick up the bunch of grapes).