Capita selecta Artificial Intelligence
High Level Robot Programming

Thijs Dieltjens Wout Vekemans

December 2015

1 Part 1

1.1 Fluents

We define fluents for the position of the robot and the positions of the minerals. The positions of the holes are fixed, and therefore not fluent.

- `currentPosition(x,y,s)`
- `mineral(x,y,s)`

1.2 Primitive Actions

The action theory to solve the Mars Rover problem contains three basic actions. Two of these are to move horizontally and vertically on the field. They each take a list that contains pairs that represent the movements. A pair that can be in this list is $[(1,2),(2,2)]$, which represents a movement to the right, starting at position $(1,2)$. The third action is one to pick up a mineral at a certain position.

- `hmove(visited,amount)`
- `vmove(visited,amount)`
- `collect_mineral((x,y))`

.

1.3 Preconditions

The preconditions for the move predicates are based on the current position, whether or not there is a hole, and whether or not the move has already been made. This last restriction is added to prevent loops. Whenever the robot arrives on a square, it should go in another direction than all previous moves from that square. The `member` that is used in the preconditions takes an element, and a list. It returns true if the element is in the list and false if the element is not in the list.

- `poss(hmove(Visited,1),s)` $\equiv$ `currentPosition(x,y,s)` $\wedge$ `x<3` $\wedge$ `x1 is x+1` $\wedge\neg$`hole((x1,y))` $\wedge\neg$`member(((x,y),(x1,y)),Visited).`
- `poss(hmove(Visited,1),s)` $\equiv$ `currentPosition(x,y,s)` $\wedge$ `x>0` $\wedge$ `x1 is x−1` $\wedge\neg$`hole((x1,y))` $\wedge\neg$`member(((x,y),(x1,y)),Visited).`
- `poss(hmove(Visited,1),s)` $\equiv$ `currentPosition(x,y,s)` $\wedge$ `y<3` $\wedge$ `y1 is y+1` $\wedge\neg$`hole((x,y1))` $\wedge\neg$`member(((x,y),(x,y1)),Visited).`
- `poss(hmove(Visited,1),s)` $\equiv$ `currentPosition(x,y,s)` $\wedge$ `y>0` $\wedge$ `y1 is y−1` $\wedge\neg$`hole((x,y1))` $\wedge\neg$`member(((x,y),(x,y1)),Visited).`

The precondition to pick up a mineral is less complex. It only states that there should be a mineral on the position the robot wants to pick up a mineral and that the robot is on that position.

- `poss(collect_mineral((x,y)),s)` $\equiv$ `currentPosition(x,y,s)` $\wedge$ `mineral((x,y),s).`

1.4 successor state Axioms

The position of the robot changes only when there is an action that is `hmove` or `vmove`

- currentPosition(x1,y,do(a,s)) $\equiv$ a = hmove(1) $\wedge$ currentPosition(x, y, s) $\wedge$ x1 = x + 1 $\vee$ $\neg\exists x, y$ a = hmove(x,y) $\wedge$ $\neg\exists x, y$ a = vmove(x,y) $\wedge$ currentPosition(x1,y,s)).
- currentPosition(x1,y,do(a,s)) $\equiv$ a = hmove(-1) $\wedge$ currentPosition(x, y, s) $\wedge$ x1 = x - 1 $\vee$ $\neg\exists x, y$ a = hmove(x,y) $\wedge$ $\neg\exists x, y$ a = vmove(x,y) $\wedge$ currentPosition(x1,y,s)).
- currentPosition(x,y1,do(a,s)) $\equiv$ a = vmove(1) $\wedge$ currentPosition(x, y, s) $\wedge$ y1 = y + 1 $\vee$ $\neg\exists x, y$ a = hmove(x,y) $\wedge$ $\neg\exists x, y$ a = vmove(x,y) $\wedge$ currentPosition(x,y1,s)).
- currentPosition(x,y1,do(a,s)) $\equiv$ a = vmove(-1) $\wedge$ currentPosition(x, y, s) $\wedge$ y1 = y - 1 $\vee$ $\neg\exists x, y$ a = hmove(x,y) $\wedge$ $\neg\exists x, y$ a = vmove(x,y) $\wedge$ currentPosition(x,y1,s)).

The `mineral` fluent only changes when the robot picks up a mineral

- mineral(position,do(a,s)) $\equiv$ mineral(position,s) $\wedge$ a $\neg$ collect_mineral(position).

1.5 Initial theory

A possible initial theory is that the robot starts in the bottom left corner of the grid, with minerals on positions (3,1) and (0,3) and a hole on position (2,2).

- `currentPosition(0,0,s0)`

- `mineral((3,0),s0)`
- `mineral((3,2),s0)`
- `hole(1,0)`

The code that implements this robot, is found in `rover.pl`. The command to run the example is `do(run, s0, S)`

2 Part 2

2.1 Omelette via Knowledge-based Programming

Our solution for the omelette problem can be found in `omelette.pl`. At the start the robot has no knowledge of which egg is good or bad. This is why he needs a sense operation which in this case is `smell(Egg)`. The robot needs this knowledge to be able to throw bad eggs away and use good ones for the omelette. The program assumes it knows a large enough number of `available(egg)`s.

2.2 Blocks on tables via Generalized Planning

To solve this problem we assume the robot has two sensing operations. The first operation senses if there are still blocks left that are either not yet considered or on the floor. The second sensing operation `senseBlock` senses if a block is either on the table (`table`), on the floor and clear (`floor`) or on the floor but unclear (`unclear`). A block variable in the program corresponds with the block we are looking at. The robot has three possible actions: move a block to the table and then go to the next block (`moveToTable`), go to the next available block if the previous block is already on the table (`nextBlock`) or skip a block and move to the following one when the previous block is unclear (`skipBlock`). The program we would like to output is the following:

```
LOOP
  CASE senseBlocksLeft OF
     -yes:
         CASE senseBlock OF
            -unclear:
                skipBlock ;
                NEXT
            -floor:
                moveToTable ;
                NEXT
            -table:
                nextBlock ;
                NEXT
         ENDC
     -no: EXIT
  ENDC
ENDL
```

The idea of our code is to have a `parm_fluent` that represents the number of blocks that are unseen or on the floor (`blocks_left`). This fluent needs to be changed whenever we move a block from the floor to the table. It also needs to be changed when we notice a previously unseen block is already on the table. When a block on the floor is unclear, this value does not need to change. Since after each action we proceed to a next block we would like to reset our initial value of block. We tried to do this with:

```
causes(ACTION,ok,block,X,or(X=unclear,or(X = floor,X=table))).
```

This only worked well for one action. For more than one action, running the program does not find a solution in acceptable time. When not considering the reset of the block value, we encountered a second problem. No solution gets found when `skipBlock` does not lower the `blocks_left` value. If we (incorrectly) do lower this value, we do get the result we want. This program can be found in `blocks1.pl`.
Since debugging did not point us to a solution, we provide a program that is suboptimal. Whenever an unclear block is considered, the value of block is changed to `table` or `floor` and the block is then processed accordingly. This could be considered as going from an unclear block to a second block that is not unclear and later the first block is reconsidered when it is no longer unclear. This code can be found in `blocks2.pl`.

2.3 Analyzing Minerals via Generalized Planning

To solve this problem we started from the provided `treechop.pl` solution because it contains a lot of similarities. We first considered the subproblem of rocks that are only covered with sand, and changed the `treechop` code in a way that the resulting program looks like this:

```
LOOP
  CASE check_sand OF
     -no: EXIT
     -yes:
         break_sand ;
         NEXT
  ENDC
ENDL
store
```

We then added a second ice layer in exactly the same way as we did with the sand layer. The first difference needed to the code is the `poss` clause for breaking the ice, since this should only be possible if the rock is no longer covered in sand. The second difference regards the `poss` clause for storing the rock. This now should only be possible when it is no longer covered with sand nor ice. The resulting code can be found in `rocks.pl`. At first running the code did not j the programs we were expecting, so we sent a first mail to clarify our problem. As a result we needed to modify Kplanner, so it would be able to work with two `parm_fluents`. After doing this, our code no longer gave any solutions. This is when we started to examine the code some more. We changed the `poss` clauses for breaking the ice so it was no longer necessary to be sand-free when breaking the ice. We then changed the goal of the code to `sand=no` or `ice=no` the expected result is returned. Whenever we change the goal to `and(sand=no,ice=no)`, a solution can no longer be found. This is to our understanding weird behaviour, since both goals in the and-expression can be satisfied on their own, without influencing the other. Thus began the next phase in trying to solve this problem, by no longer using two `parm-fluents` but mimic them using only one. We tried the following code as it was suggested:

```
parm_fluent((ice_max,sand_max)).
init_parm(generate,(ice_max,sand_max),(1,1)).
init_parm(test,(ice_max,sand_max),(50,50)).
init(ice_max,A) :- init((ice_max,sand_max),(A,_)).
init(sand_max,A) :- init((ice_max,sand_max),(_,A)).
```

This again did not provide any results, and finished even more quickly. At this point we already spent more than 15 hours on this problem, so we stopped trying to get a solution because of these unexplainable bugs.