Homework 1 - PAR-MAI

Jordi Armengol

October 2019

1. Define two very different ways to represent the lunar lockout world (objects, states, actions, domain axioms). Ensure there is a big difference in the set of possible states between both representations.

   Answer:

   Option 1: The general idea of this option is to look for a pair of crafts such that we can move one of them into the other one, and then directly move the first craft into the other one.

   - **Types**: Craft and Coordinate.
   - **Objects:** Crafts: redCraft greenCraft purpleCraft yellowCraft orangeCraft. Coordinates: zeroCoord oneCoord twoCoord threeCoord fourCoord.
   - **States (predicates):** The states are composed of the combinations of the following predicates: the location of the craft, whether a coordinate is less than another one, and whether a coordinate is exactly one less than another one. In practice, since these two last predicates are constant, the states will consist of the different positions of the crafts, which can be positioned everywhere as long as they are in different positions.
   - **Actions:** (go) up, down, left, and right (the four possible movements of the crafts). This is detailed in the PDDL code, but notice that since the general idea is to look for a pair of crafts such that we can move one of them into the other one, we are going to take two crafts as parameters (not only one), as well as their respective positions and the future position of the craft that we are going to move. Crucially, one of the two current coordinates of the two crafts will be parameterized as a single parameter (eg. for moving left, we are going to take a shared Y coordinate for both crafts, since if we want to move one craft into another by moving left they should have the same Y coordinate), forcing the planner to not take into account actions that would not be feasible from the very beginning, instead of checking it in the pre-condition.

- **Domain axioms:** They are implicit in the pre-conditions and effects of the actions (defined later on in PDDL code), but the main ones are:
  - A craft can only be located at one position (coordinate x, coordinate y) at the same time.
  - A craft cannot be located at the same location as another one.
  - A craft cannot *jump over* another craft.
  - A craft can only move horizontally or vertically.
  - A craft cannot hit borders (ie. a craft can only move towards another craft, and in particular immediately next to this other craft).
- **Problem instances**: All problem specifications require to declare the aforementioned objects and the less and exactly less than one relationships, since PDDL has no inherent notion of arithmetic (unless we use the fluents extension). In addition, the initial positions of the different crafts have to be given as well.

Option 2: Alternatively, instead of directly trying to hit one craft into another one, we could have more fine-grained movements and go cell by cell. Notice that this will be less efficient and the number of states will increase.

The relevant differences with respect to the previous solution are marked in blue.

- **Types**: Craft, Coordinate, Movement.
- **Objects:** Crafts: redCraft greenCraft purpleCraft yellowCraft orangeCraft. Coordinates: zeroCoord oneCoord twoCoord threeCoord fourCoord. Movements: upMov, downMov, leftMov, rightMov, noMov.
- **States (predicates):** The states are composed of the combinations of the following predicates: the location of the craft, whether a coordinate is less than another one, whether a coordinate is exactly one less than another one, and whether a craft is doing a certain kind of movement. In practice, since the two predicates for arithmetic comparisons are constant, the states will consist of the different positions of the crafts, which can be positioned everywhere as long as they are in different position, and the kind of movement that each one is doing.
- **Actions:** (go) up, down, left, and right (the four possible movements of the crafts). Notice that since the general idea is to look for a pair of crafts such that we can move one of them into the other one, we are going to take two crafts as parameters (not only one), as well as their respective positions and the future position of the craft that we are going to move. Crucially, one of the two current coordinates of

the two crafts will be parameterized as a single parameter (eg. for moving left, we are going to take a shared Y coordinate for both crafts, since if we want to move one craft into another by moving left they should have the same Y coordinate), forcing the planner to not take into account actions that would not be feasible from the very beginning, instead of checking it in the pre-condition. The difference with respect to the previous solution is that now we will have to check whether there exists a craft that is moving. In order not to break the rules of the game, if a craft is already moving, we will have to continue the same movement (eg. if the red craft started going up in the previous step, in the following movement the same craft will have to go up as well). Apart from the pre-condition, with regard to the effect, both the *add list* and the *delete list* will be modified with respect to the previous solution. In this case, if no craft is moving (ie. all of them have the predicate *isMoving* to *noMov*), then we can choose a craft to change its movement kind to a certain direction. Once the craft that is moving is immediately next to the other craft, then we have to set its respective *isMoving* predicate to *noMov* again.

- **Domain axioms:** The main ones are:
  - A craft can only be located at one position (coordinate x, coordinate y) at the same time.
  - A craft cannot be located at the same location as another one.
  - A craft cannot *jump over* another craft.
  - A craft can only move horizontally or vertically.
  - A craft cannot hit borders (ie. a craft can only move towards another craft, and in particular immediately next to this other craft).
  - Only one craft can be moving at the same time. This was still true in the previous solution, because the game rules are the same, but in this case the axiom must be explicit in the pre-conditions and effects. In the previous solution, since the crafts moved directly to their new assigned position, the intermediate states of the craft moving towards its new position did not exist.
- **Problem instances**: All problem specifications require to declare the aforementioned objects and the less and exactly less than one relationships, since PDDL has no inherent notion of arithmetic (unless we use the fluents extension). In addition, the initial positions of the different crafts have to be given as well. Moreover, initially, all crafts must be set to null movement.

2. Explain the number of possible states for both representations that you came up with. Do they also differ in the number of feasible states? Explain why each representation would be preferable to the other.

   Answer:

Option 1: States are defined by predicates, so we should compute the number of combinations of the different predicates. However, the two predicates for defining arithmetic relations are constant (eg. oneCoord is *always* defined to be less than *threeCoord*, and this never changes). Thus, they should not be factored in the computation (ie. we are not going to take into account alternatives definitions of arithmetic in which, for instance, one is bigger than two, even though such scenario could be possible with the domain file alone). The number of possible states is actually defined by the predicate defining the positions of the crafts. Naively, this could be approximated as:

$$\#states = (\#coords \times \#coords\times)^{\#crafts} = (5 \times 5)^5 = 9765625$$

However, not all these positions are feasible, since no crafts can be located at the same position. We have no restrictions to position the first craft, we can position in all but one cell the second craft (the one of the first craft), and so on. Therefore:

$$\#states = 25 \times 24 \times 23 \times 22 \times 21 = 6375600$$

We cannot further cut down the number of possible states, because the game has no rule stating that initial states must follow a certain restriction (eg. there is no rule stating that initial states should always lead to games that can be solved).

Option 2: The same we have said for the previous case applies to the second solution. However, this time we will have to keep track of the current movements of the crafts, since we are moving cell by cell. Therefore, in addition to all the possible configurations of the board, now we have to factor in the current movements and this could be approximated as follows:

$$\#states = 25 \times 24 \times 23 \times 22 \times 21 \times \#movements \times \#crafts$$

$$25 \times 24 \times 23 \times 22 \times 21 \times 5 \times 5 = 159390000$$

which is considerably bigger than the number of possible states of the possible solution. However, since only one craft can be moving at the same time, we don't have to take into account all the states of movement for all crafts, but the 4 directions for all crafts plus the state in which of all them are stopped. The number of possible states is actually:

$$\#states = 25 \times 24 \times 23 \times 22 \times 21 \times (5 \times 4 + 1) = 133887600$$

which is still considerably bigger than the result of the previous solution.

So, the two solutions differ in both the number of the coarse-grained approximation of states and the number feasible states.

At least with these rules, the first solution is clearly preferable because it has a way lower number of states and therefore is more efficient. Since once

a craft starts moving it cannot stop until it hits another craft (according to the game rules), it makes no sense to have the level of movement granularity of the second solution (cell by cell). However, the second solution is more flexible and general, and could be easily adapted to similar games with slightly different rules.

3. Write the PDDL files required for a planner of your choice to solve this problem (Hint: a statespace search is likely the most simple). We should be able to run your code with something like the following command: `./lunar-lockout <domain.pddl> <problem.pddl>`. The `<domain.pddl>` should define the lunar lockout world actions and predicates. If you wish, you can hardcode the domain representation into your suggested planner but be sure to state this in your README. The ¡problem.pddl¿ contains the objects, init state, and goal state. The output of your planner should be a sequence of actions that solves the given puzzles. Explain how we should interpret the output. It is OK to use a more compact representation.

   Answer: The first option, the preferable solution, is the one that has been implemented. As attached files, we have a domain file, `domain.pddl`, and two problem files, `problem1.pddl` and `problem2.pddl`, corresponding to the examples in the assignment. Notice that a script for automatically generating problems have been implemented as well, in *problem_gen.py*.

   For generating plans, just run a planner such as FF or an online service such as this one specifying the respective domain and problem files.

   The output of the first problem is as follows:

```
 Match tree built with 246 nodes.

PDDL problem description loaded:
Domain: LUNAR_LOCKOUT
Problem: PROBLEM1
#Actions: 246
#Fluents: 48
Landmarks found: 1
Starting search with IW (time budget is 60 secs)...
rel_plan size: 3
#RP_fluents 5
Caption
{#goals, #UNnachieved,  #Achieved} -> IW(max_w)

{1/1/0}:IW(1) -> [2][3][4][5]rel_plan size: 0
#RP_fluents 0Plan found with cost: 4
Total time: -7.39098e-09
Nodes generated during search: 17
Nodes expanded during search: 13
```

```
IW search completed
0.00100: (up redcraft fourcoord fourcoord zerocoord onecoord orangecraft)
0.00200: (left redcraft onecoord fourcoord twocoord threecoord greencraft)
0.00300: (down redcraft threecoord onecoord threecoord twocoord yellowcraft)
0.00400: (left redcraft twocoord threecoord onecoord twocoord purplecraft)
Planner found 1 plan(s) in 0.737secs.
```

The output is quite verbose, but easy to interpret. The relevant information are the last lines, which outline the steps of the generated plan. For instance, `(up redcraft fourcoord fourcoord zerocoord onecoord orangecraft)` means that the redcraft must move up. The rest are just the parameters that were used in the action (in this case: the red craft was located at (4,4), the orange craft, the one we want to move into, is located at (4,0), and the new position of the red craft will be (4,1)). So, our planner suggests:

(a) Move red craft up.

(b) Move red craft left.

(c) Move red craft down.

(d) Move red craft left.

which is a correct solution to the first example.

In the case of the second example:

```
Planning service: http://solver.planning.domains/solve
Domain: lunar_lockout, Problem: problem2
 --- OK.
 Match tree built with 644 nodes.

PDDL problem description loaded:
Domain: LUNAR_LOCKOUT
Problem: PROBLEM2
#Actions: 644
#Fluents: 78
Landmarks found: 1
Starting search with IW (time budget is 60 secs)...
rel_plan size: 4
#RP_fluents 8
Caption
{#goals, #UNnachieved,  #Achieved} -> IW(max_w)

{1/1/0}:IW(1) -> [2][3][4][5]rel_plan size: 0
#RP_fluents 0Plan found with cost: 4
Total time: 4.05312e-09
Nodes generated during search: 42
Nodes expanded during search: 24
```

```
IW search completed
0.00100: (down purplecraft onecoord onecoord fourcoord threecoord orangecraft)
0.00200: (right purplecraft threecoord onecoord threecoord twocoord greencraft)
0.00300: (right redcraft zerocoord zerocoord threecoord twocoord yellowcraft)
0.00400: (down redcraft twocoord zerocoord threecoord twocoord purplecraft)
Planner found 1 plan(s) in 0.726secs
```

which means:

(a) Move purple craft down.

(b) Move purple craft right.

(c) Move red craft right.

(d) Move red craft down.

which is a correct solution to the second example.

For generating new problems, the usage of the script is:

```
python3.6 <problem_name> [craft x y] > problem_name.pddl
```

with craft in [red, green, purple, yellow, orange] and x, y in [0-4] such that all crafts are defined. For instance:

```
python3.6 problem_gen.py problem5 orange 4 0 green 2 1
    purple 1 2 yellow 3 3 red 4 4 > problem5.pddl
```

The script can generate random instances as well:

```
python3.6 <problem_name> random <seed> > problem_name.pddl
```

For instance:

```
python3.6 problem_gen.py problem5 random 1 > problem5.pddl
```

Notice that in the case of random generations it is guaranteed that all the generated problems will be valid (because the script checks whether two crafts are in the same position), but it is not guaranteed that the problem will have a solution.