Lunar Lockout

Josep de Cid

October 2019

1 Introduction

Lunar Lockout Game is a board game consisting in a 5x5 board with several spacecrafts. The objective is to move the Red spacecraft to the central cell, marked as red. In each turn, a spacecraft moves in a specific direction, and stops when it collides with another spacecraft. It is illegal for any of the spacecrafts to leave the board, so there always must be a spacecraft that blocks the moving spacecraft in that direction.

2 Domain Representation A

The first representation presented, is the implemented in the `pddl` files attached. It is the simplest and most efficient and its summary can be seen in table 1.

Objects	Predicates (States)	Actions
R, O, Y, G, B, P	at, empty	move-up, move-down
C1, C2, C3, C4, C5	next, adjacent	move-left, move-right

Table 1: Summary of Objects, Predicates and Actions for Domain A

The main idea is to given an spacecraft, find a cell in the same row or column where it can be moved if there is an spacecraft that can block its movement.

Objects

There are two different objects in our problem, the Spacecrafts (`spacecraft`) and Coordinates (`coord`), inheriting from `object`. In the domain file we define these types:

```
(:types coord - object
        spacecraft - object)
```

and in each problem file, we create the instances of these objects:

```
(:objects Red Orange Yellow Green Blue Purple - spacecraft
          C1 C2 C3 C4 C5 - coord)
```

The choice to represent the board with a single dimension of coordinates is due to the redundancy to repeat objects for x and y, as they both have the same behaviour. We will define how the cells are positioned with respect to the others using predicates, emulating arithmetic relations between indices $1 < 2 < \ldots < 5$, and also defining if two indices are adjacent.

We don't really need to differentiate between the Red spacecraft and the Helpers as all of them can perform the same actions, however, if for instance only the Red one would be able to move, we could create two subclasses of `spacecraft` such as `red` and `helper`, and set the spacecraft in the actions parameters to be `red` type.

Predicates

We define 4 different predicates, two for spacecraft position checking and two for movement purposes:

```
(at ?s - spacecraft ?x ?y - coord)
```

The predicate receives an spacecraft and a pair of coordinates *(x, y)*. It is true if the spacecraft s is located at (x, y).

```
(empty ?x ?y - coord)
```

Checks the emptyiness of a cell *(x, y)*. Despite not being a mandatory predicate, it is far more efficient to have a predicate to check if a cell is empty rather than using `forall` or `exists` statements to check that there is no spacecraft in the given position.

```
(next ?x ?y - coord)
```

Indicates that the coordinate y is after x (the index is greater). It is useful to check if, fixing one dimension, if one spacecraft is above (or any other direction) another given their coordinates in the other dimension.

```
(adjacent ?x ?y - coord)
```

Indicates that the coordinate y is inmediately after x (the index y - 1 is the index x). It's used to check if given the target coordinate there is a spacecraft in the given direction adjacent to that coordinate.

Functions, Metrics and Goals

We propose a single numeric function, `movements`, which counts the movements needed to reach the plan goal, to position the red spacecraft in the central cell

of the board. It is initialized as 0 with `(= (movements) 0)` in the problem files, and is increased by 1 as an effect of any action with `(increase (movements) 1)`.

This is used later in the problem metrics to `minimize` the number of movements required to complete the problem and obtain the cheapest solution. The goal is to reach the central position with the Red spacecraft:

```
(:goal (at Red C3 C3))
(:metric minimize (movements))
```

Actions

The first domain has 4 actions, `move-x` to each direction, being `x top`, `left`, `bottom` and `right`. These actions could be somehow simplified into a single one, but for readability and efficiency terms, it is better to keep them as different ones. We will describe just one, *e.g.* `move-up`, as all the others are symmetric with respect to the others.

Parameters

The function receives four parameters, the spacecraft, it's coordinates and one of the target cell coordinates. In previous versions, both target cell coordinates were given as parameters, but as the spacecraft can only be moved straight, the other coordinate will remain the same, being unnecessary to pass both as parameters. For `up` and `down` directions the x is modified, and for `left` and `right` we receive the new y.

There are multiple variations for the parameters to receive, such as passing the blocking spacecraft, both coordinates... but for terms of simplicity and interpretability of the resulting plan, we have decided this option as it shows the spacecraft in its current state and where is it going, without unnecessary information such as which other spacecraft blocked it.

```
:parameters (?ship - spacecraft ?x ?y ?xtarget - coord)
```

Precondition

In the precondition we must start checking some basic conditions to be satisfied, such as than the spacecraft is at the given positions in the parameters, the target cell is empty (same x and target y) and that the target cell is above the current one. These checks are set at the beginning for terms of efficiency, to avoid checking first the following `exists` and `forall` conditions.

```
(at ?ship ?x ?y)
(empty ?xtarget ?y)
(next ?xtarget ?x)
```

We also must check that there is a blocking spacecraft on top of the target cell, adjacent to it, so we use an `exists` clause to check if there is an spacecraft at the cell c' such that c' is right above of c, the target cell:

```
(exists (?limit - spacecraft ?xlimit - coord) (and
    (at ?limit ?xlimit ?y)
    (adjacent ?xlimit ?xtarget)
))
```

Last precondition to check is that there are no obstacles between the current cell and the target one. In order to do that, we use a `forall` clause, and check than there is no other spacecraft between the target and the current one:

```
(forall (?obstacle - spacecraft ?xobstacle - coord)
    (not (and
        (not (= ?ship ?obstacle))
        (at ?obstacle ?xobstacle ?y)
        (next ?xtarget ?xobstacle)
        (not (next ?x ?xobstacle))
    ))
)
```

Effects

The main effect is that the spacecraft is not anymore at the original position and now it is at the target one. Hence we also must update the original to be empty and remove the emptiness of the new one. As commented in the functions section, we increase the number of movements by one:

```
:effect (and
    (not (at ?ship ?x ?y)) (empty ?x ?y)
    (at ?ship ?xtarget ?y) (not (empty ?xtarget ?y))
    (increase (movements) 1)
)
```

States

The state space is all the possible environments for the problem, *i.e.* all possible combination of predicates in our domain. Given the difficulty to provide an exact value for the number of states, we will use Big O notation, and then refine it a bit giving the number of feasible states.

As we have at most 6 spacecrafts and exactly 25 cells, the predicate `at` can create 150 different states. The predicate `empty` adds 25 more to the count (number of cells), and `next` and `adjacent` admit 20 possible values for each one. However, this number is so easy to cut down.

The way that `next` and `adjacent` predicates have been defined and what they represent is problem independent, so there's always the same number of `true` predicates of this kind, 10 for `next` and 4 for `adjacent`.

The same happens with the other two predicates, two spacecrafts can't overlap, and one cell can be only occupied xor empty at the same time. Hence, the predicate `at` for the first spacecraft can take 25 values, the second 24, and so on. On the other hand, `empty` is like a complementary of `at`, so it doesn't really provide a new set of states, being the result $25 * 24 * 23 * 22 * 21$.

3 Domain Representation B

The second representation, is worse than the first version (section 2). To avoid explaining in detail everything again and being redundant, we comment just the differences such as new predicates and small changes in the actions, as well as describing the states of this representation. The summary of this version is shown in table 2

Objects	Predicates (States)	Actions
R, O, Y, G, B, P C1, C2, C3, C4, C5 U, D, L, R	at, empty next, adjacent moving	move-up, move-down move-left, move-right

Table 2: Summary of Objects, Predicates and Actions for Domain B

This version, moves the spacecraft cell by cell, adding a new predicate that indicates that it is moving and in which direction. Hence, we must also include new objects to represent the directions, such as numbers $\{0, \ldots, 3\}$ for *top*, *right*, *bottom* and *right* respectively.

```
(moving ?s - spacecraft ?d - direction)
```

Each move action, must include in the precondition a check that the spacecraft is not moving in any other direction, and that no other spacecraft is moving. The effects set this predicate `moving` to `true` until it is blocked by another spacecraft, then it sets the predicate to `false`, allowing it to move in another direction.

States

The number of states produced by this version is much higher, as it adds the possibility for each spacecraft to be moving in any of the four directions (not in terms of feasibility, as at most one spacecraft and one direction must satisfy the `moving` predicate. Hence, this is a worse representation than the previous one, as it adds unnecessary states that can be avoided just moving the spacecraft directly to the target position checking if it is a valid move, multiplying by 4 the previosu result.

Furthermore, the efficiency of the planner decreases so much with this version, as a lot of actions could be performed just to be discarded later as *e.g.* there is no blocking spacecraft and one spacecraft has been moving up 4 times in a row. It also adds some `exists` or `forall` statements, which we must avoid as much as we can, for efficiency terms.

4 Execution and Evaluation

To execute the planner with our domain we will need to have installed `METRIC-FF`. The command to run the planner is:

```
> ./ff -o lunar_lockout_domain.pddl -f lunar_lockout_X.pddl
```

where `X` is the number of the problem (from 1 to 40). However, during the development we've used a Visual Studio Code plugin PDDL by Jan Dolejsi which uses an online API planner. This is an example of the output:

```
(move-left green c1 c3 c2)  ; Green from  (1,3) to (1,2)
(move-left purple c1 c5 c3) ; Purple from (1,5) to (1,3)
(move-down green c1 c2 c4)  ; Green from  (1,2) to (4,2)
```

To ease the process of defining different problems to test the domain, we provide a Python script[1] to create the initialization of the objects dynamically.

To create a new problem it's only necessary to adjust value of the variables `problem_number` and `spacecrafts`, that contain the coordinates $\in \{1, \ldots, 5\}$.

```
problem_number = 40
spacecrafts = { 'Red': (5, 2), 'Orange': (1, 1), 'Yellow': (4, 5),
                'Green':  (1, 3), 'Blue':   (5, 5), 'Purple': (1, 5) }
```

The script creates the specific *init* section of the problem, with all the predicates `at` and `empty`, and setting the `problem_number` in problem and file names.

To evaluate the problem we also provide another script[2] which creates a board visualization using TkInter library, the standard GUI package for Python. Reading the actions provided as the result of the planner execution, it replays the movements to see visually the actions performed (example in figure 1).

This script works only with the output format provided by the FF-Metrics planner. As an example, to avoid problems without the proper environment, the planner output for the problem #40 is attached in `plan.txt`, use the `--file` option in the script. We also provide a few already codified puzzles to test, #1 and #2 (defined in the statement), and four of the considered as expert level (from #37 to #40).

[1] `generator.py` attached in the zip file

[2] `visualizer.py` attached in the zip file

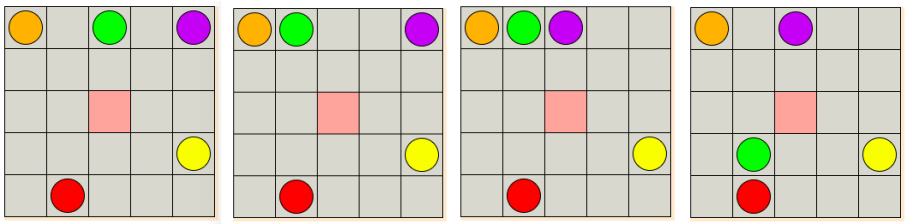

Figure 1: First steps of the visualization for Problem 40

5 Conclusions

PDDL is a good language to face this kind of problems, as it provides a general way to codify them and then, all the hard job is done internally by the planner, such as `FF`.

It's quite easy to find a general state-like representation for this problem. The chosen representation is suitable for this problem, however it doesn't scale quite well with general $N \times N$ board games where, for instance, it would be more appropriate to use numerical values for cells, using *fluents*.