Homework 1 : Lunar-lockout game

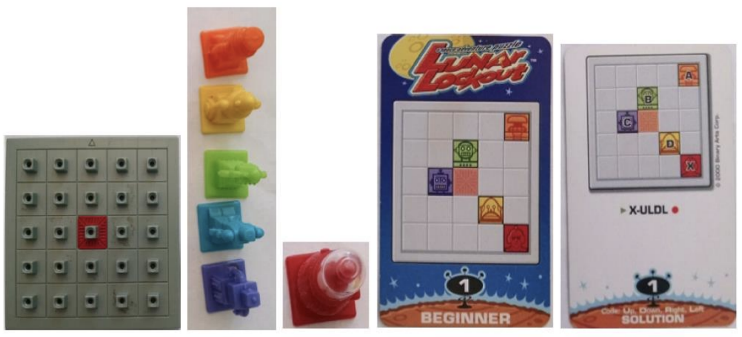
Jana Reventós Presmanes

MAI - Planning - October 2019

1 Introduction

The Lunar - lockout game is a space adventure puzzle in which you have to use five loyal helper spacecrafts to find a way back to place the red spacecraft into the ship's emergency entry port. Spacecrafts can only moves in straight lines (up-down or left-right) and as well, can only move a direction if there is another spacecraft in the way to avoid careening into the space. Therefore, all spacecrafts need to cooperate to stay on the board and help the red spacecraft to reach the goal.

This solitary game consist of 6 spacecrafts players (1 red and 5 helpers spacecraft), a 5x5 game board with a red square in the center (emergency entry port) and different initial cards that show the spacecrafts starting positions of the puzzle. Cards range from beginner, to intermediate, to advance and expert levels.



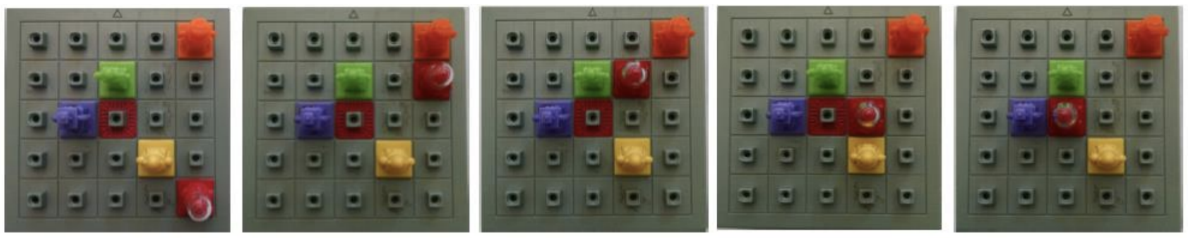
(a) 5x5 Board (b) Helpers (c) Red (d) Puzzle 1 (e) Solution 1

Figure 1: Lunar-lockout game pieces

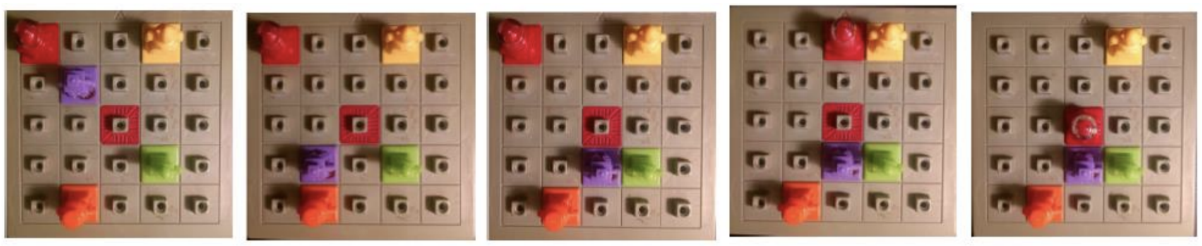
In this laboratory work we have used the Planning Domain Definition Language (PDDL) in order to develop a code which finds a right plan to achieve the game objectives keeping in mind all the described rules. It is important to remind that the PDDL code is always divided in two files, the domain and the problem. The domain.pddl file contains predicates and actions whereas the problem.pddl document includes objects, initial states and goal specifications. The plan lunched by the designed code outputs the different moves that the spacecrafts have to do in order to reach the goal.

2 Objectives

The main objective of this labwork is to create a pddl code able to solve two different puzzles of a typical game play. In puzzle 1 the solution only requires moving the red spacecraft while helpers remain at their initial position as seen in Figure 2 top. On the other hand, puzzle 2 requires cooperation between all spacecrafts in order to reach the desired goal, as shown in Figure 2 bottom. It is important to mention that the initial position of each spacecraft will be defined in the problem.pddl file, so for each different puzzle card a new problem.pddl has to be written.



(a) Init (b) Move up (c) Move Left (d) Move Down (e) Move Left



(a) Init (b) P - Move Down (c) P - Move Right (d) R - Move Right (e) R - Move Down

Figure 2: Puzzle 1 and puzzle 2 cards examples

3 Questions

a) Define two very different ways to represent the lunar lockout world (objects, states, actions,domain axioms). b) 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.

The first approach found to represent the lunar lockout world is the one represented in Table 1. The planning tasks are divided in two files: states and actions in the domain file and objects, initial conditions and game goal in the problem file. It is important to state that in this solution the board squares (5x5) are created using columns or row localization from s1 to s5. For instance, the red square in the center of the table will have a localization of s3 s3 (col 3, row 3). The states or predicates define the properties of the objects that we are interested in. In this example, "spacecraft at" will indicate the localization of the red or helper spacecrafts within the board game, "bigger" predicate shows us whether one column or row has a bigger number, "lower" says that a column or row has lower associated number and "adj unit greater" denotes that a column or row is only one unit greater than the next one therefore, there are adjacent. Finally, the goal is to put the red spacecraft at the center of the board.

First approach			
Objects	States	Actions	Goal
spacecraft	spacecraft at	move right	red at center
locations	bigger	move left	
	lower	move down	
	adj unit greater	move up	

Table 1: Objects, states, actions and goal defined in the pddl files of the first approach

This second approach represented in Table 2 contains different objects, states and actions as the fist one. First, is important to mention than the idea behind this solution is built in a game board of 5x5 where each square is an object. For example, the central localization of the table will be expressed as s-3-3 where the first number is the column and the second the row, so a total of 25 object must be described. In this defined case, the spacecraft object are divided in two: spacecraft (which is the red one) and helpers (the other 5 game players). Furthermore, actions have shrunk as we only define the "straight move" action. This action will indicate that the red or helper spacecraft can only move in a straight direction where another piece is. The goal is the same as before.

Second approach			
Objects	States	Actions	Goal
spacecraft	spacecraft at	straights move	red at center
helpers	helper at		
locations	adjacent		

Table 2: Objects, states, actions and goal defined in the pddl files of the second approach

The number of feasible state do differ between both approaches due the different board representation described in each solution. In the first approach "bigger" and "lower" states must be established as the same coordinate for a column or row is not enough information to build the board game, remember that in this case we only have 5 objects for the locations. For instance, in the second approach each square is expressed as a location, so only adjacency has to be defined.

The first representation would be preferable to the second one because we won't need to define 25 objects for the board locations, so the development of the initial conditions would be easier and faster by using bigger, lower and adjacent and one unit greater states. If the second representation is used the adjacency between pairs of table square must be defined twice, e.g. adj s-1-1 s-1-2 and adj s-1-2 s-1-1, which is an onerous task and less efficient. On the other hand, the four movement actions represented in the first approach can be replaced for just one movement as in the second representation because a straight movement could be achieved moving the red or helper spacecraft square by square. However, the design of this movement would need more preconditions in comparison with defining separate movements (right-left and up-down). In conclusion, the first approach seems to be an easy way to built a pddl code that plans a solution for the given puzzles.

c) Write the PDDL files required for a planner of your choice to solve this problem
The planner that has been chosen to solve this problem is the one described in Table 1. This planner contains one domain and three problems, two problems which solve the beginner-puzzles explained above and one extra problem which includes the initial conditions for a puzzle of an expert level.

Domain explanation

The domain pddl file contains predicates and actions as well as the optional commands of requirements and types. This domain is able to solve all the game puzzles, from beginner level to expert level. In other words, it can be used to solve those puzzles where helper spacecrafts remain in their initial positions and as well, those puzzles where cooperation between all spacecrafts is needed in order to achieve the goal.

- Requirements:
:strips
Stanford Research Institute Problem Solver (STRIPS) is an automated planner.
:typing
This requirement means that the domain uses types.
:adl
Action Description Language (ADL) is an automated planning and scheduling system.
:fluents
To specify action costs, it keeps track of the cost.

- Types: syntax for declaring parameters and objects, "spacecraft" and "locations" are declared before being used in the domain

- Predicates: are described in the state column in Table 1.
1) `spacecraft_at ?red_or_helper ?col ?row` indicates whether the red spacecraft or a helper spacecraft is located at specific column and row.
2) `bigger ?colrow1 ?colrow2` says that the first column/row has a bigger value than the second one.
3) `lower ?colrow1 ?colrow2` says that the first column/row has a lower value than the second one.
4)`adj_unit_greater ?colrow1 ?colrow2` states that the first column/row is one unit greater than the second one, therefore there are adjacent in the board game.

- Functions: Just one function named `total_moves` is defined for counting the total number of times that an action is used in this designed planner.

- Actions: a total of four actions are described. Each action has two main characters, the first one is the moving spacecraft (`moving_sc`) and the second one the static spacecraft (`static_sc`). The moving one is the character that changes its initial position in the same direction of the static spacecraft. For the `move_right` and `move_left` actions the row (`row`) keeps the same during all the execution whereas the column for the moving spacecraft changes, `from_c - to_c` and the static spacecraft remains at the same column, `static_c`. On the other hand, for the `move_down` and `move_up` actions the column (`col`) keeps the same during all the execution whereas the row for the moving spacecraft changes, `from_r - to_r` and the static spacecraft remains at the same row, `static_r`.

The preconditions describe the different states of the objects before the action effect. They specify which are the locations of the moving and static spacecrafts in the four directions and they say that there isn't any other spacecraft in the moving direction. Finally, the effect of the actions is always the same, to change the initial position of the moving spacecraft to the one adjacent to the static spacecraft.

In addition, for each move in a specific direction the `total_moves` function increases one unit in order to finally count the total action cost of this planner.

Listing 1: Domain pddl specifications

```
(define (domain lunar_lockout_game)
    (:requirements :strips :typing :adl :fluents )

    (:types locations spacecraft)

(:predicates
        (spacecraft_at ?red_or_helper -spacecraft ?col - locations ?row -locations)
        (bigger ?colrow1 - locations ?colrrow2 - locations)
        (lower ?colrow1 - locations ?colrow2 - locations)
        (adj_unit_greater ?colrow1 - locations ?colrow2 -locations))

(:functions (total_moves))

(:action move_right
    :parameters (?sc_moves ?sc_static - spacecraft
                 ?row ?from_c ?to_c ?static_c - locations)

    :precondition (and
                       (spacecraft_at ?sc_moves ?from_c ?row)
                       (spacecraft_at ?sc_static ?static_c ?row)
                       (bigger ?to_c ?from_c)
                       (bigger ?static_c ?to_c)
                       (adj_unit_greater ?static_c ?to_c)
                       (not(and (spacecraft_at ?sc_moves ?to_c ?row)
                            (spacecraft_at ?sc_static ?to_c ?row)))
                        (forall (?sc_others - spacecraft ?col - locations)
                               (not (and (spacecraft_at ?sc_others ?col ?row)
                                         (bigger ?col ?from_c)
                                         (bigger ?to_c ?col)))))

    :effect(and (spacecraft_at ?sc_moves ?to_c ?row)
                (not(spacecraft_at ?sc_moves ?from_c ?row))
                (increase (total_moves) 1)))
```

```
(:action move_left
    :parameters (?sc_moves ?sc_static - spacecraft
                ?row ?from_c ?to_c ?static_c - locations)

    :precondition (and
                        (spacecraft_at ?sc_moves ?from_c ?row)
                        (spacecraft_at ?sc_static ?static_c ?row)
                        (lower ?to_c ?from_c)
                        (lower ?static_c ?to_c)
                        (adj_unit_greater ?to_c ?static_c)
                        (not(and (spacecraft_at ?sc_moves ?to_c ?row)
                                 (spacecraft_at ?sc_static ?to_c ?row)))
                         (forall (?sc_others - spacecraft ?col - locations)
                                 (not (and (spacecraft_at ?sc_others ?col ?row)
                                       (bigger ?from_c ?col )
                                       (bigger ?col ?to_c)))))
    :effect(and (spacecraft_at ?sc_moves ?to_c ?row)
                (not(spacecraft_at ?sc_moves ?from_c ?row))
                (increase (total_moves) 1 )))

(:action move_down
    :parameters (?sc_moves ?sc_static - spacecraft
                 ?col ?from_r ?to_r ?static_r - locations)

    :precondition (and
                        (spacecraft_at ?sc_moves ?col ?from_r)
                        (spacecraft_at ?sc_static ?col ?static_r)
                        (bigger ?to_r ?from_r)
                        (bigger ?static_r ?to_r)
                        (adj_unit_greater ??static_r ?to_r)
                        (not(and (spacecraft_at ?sc_moves ?col ?to_r)
                             (spacecraft_at ?sc_static ?col ?to_r)))
                         (forall (?sc_others - spacecraft ?row - locations)
                                 (not (and (spacecraft_at ?sc_others ?col ?row)
                                           (bigger ?row ?from_r)
                                           (bigger ?to_r ?row)))))

    :effect(and (spacecraft_at ?sc_moves ?col ?to_r)
                (not(spacecraft_at ?sc_moves ?col ?from_r))
                (increase (total_moves) 1)))

(:action move_up
    :parameters (?sc_moves ?sc_static - spacecraft
                 ?col ?from_r ?to_r ?static_r - locations)

    :precondition (and
                        (spacecraft_at ?sc_moves ?col ?from_r)
                        (spacecraft_at ?sc_static ?col ?static_r)
                        (lower ?to_r ?from_r)
                        (lower ?static_r ?to_r)
                        (adj_unit_greater ?to_r ?static_r)
                        (not(and (spacecraft_at ?sc_moves ?col ?to_r)
                             (spacecraft_at ?sc_static ?col ?to_r)))

                        (forall (?sc_others - spacecraft ?row - locations)
                                 (not (and (spacecraft_at ?sc_others ?col ?row)
                                           (bigger ?from_r ?row)
                                           (bigger ?row ?to_r)))))
    :effect(and (spacecraft_at ?sc_moves ?col ?to_r)
                (not(spacecraft_at ?sc_moves ?col ?from_r))
                (increase (total_moves) 1)))
)
```

Problems explanation

The problem definition contains the objects which appear in the problem, the initial conditions of the objects and the game goal. A total of three problems are being designed to solve three different puzzles as mentioned before. All problem pddl files have the same objects and final goal. Only two type of object need to be specified, spacecraft and locations. For the spacecraft type 6 object are defined: `red orange yellow green blue purple` and for the location type 5 objects are defined: `s1 s2 s3 s4 s5` which indicates the column or row number. The goal is always to place the red spacecraft at the center position of the 5x5 board game.

The list of initial conditions describe all the predicates that are true in the initial state. These conditions that are described only differ between the purposed problems in the initial location of each spacecraft. The 5x5 table is defined using the `bigger`, `lower` and `adj_unit_greater` predicates as seen in Listing 2.

In addition, the `:metric` state is used to minimize the cost of the plan. By using this statement the problem is forced to search for the plan than requires less moves (actions).

The following problem pddl example contains the initial conditions for the puzzle 2 card.

Listing 2: Domain pddl specifications

```
(define (problem puzzle_2)
    (:domain lunar_lockout_game)
(:objects
    ; the red spacecraft & helper spacecrafts objects
    red orange yellow green blue purple - spacecraft

    ; Number of column or row
    s1 s2 s3 s4 s5  - locations)

(:init  ; PUZZLE 2

    ;initial position spacecraft objects
    (spacecraft_at red s1 s1)
    (spacecraft_at orange s2 s5) (spacecraft_at yellow s4 s1)
    (spacecraft_at green s4 s4) (spacecraft_at purple s2 s2)

    ; Columns or rows that they are adjacent and one unit greater
    (adj_unit_greater s5 s4)
    (adj_unit_greater s4 s3)
    (adj_unit_greater s3 s2)
    (adj_unit_greater s2 s1)

    ; Grid 5x5
    ; we impose which columns or rows are bigger than the others
    (bigger s2 s1) (bigger s3 s1) (bigger s4 s1) (bigger s5 s1)    ; for the s1
    (bigger s3 s2) (bigger s4 s2) (bigger s5 s2)                   ; for the s2
    (bigger s4 s3) (bigger s5 s3)                                  ; for the s3
    (bigger s5 s4)                                                 ; for the s4
    ; we impose which columns or rows are lower than the others
    (lower s1 s2) (lower s1 s3) (lower s1 s4) (lower s1 s5)        ; for the s1
    (lower s2 s3) (lower s2 s4) (lower s2 s5)                      ; for the s2
    (lower s3 s4) (lower s3 s5)                                    ; for the s3
    (lower s4 s5)                                                  ; for the s4


    (= (total_moves) 0))

(:metric minimize(total_moves))

; GOAL: the red spacecraft on the middle square
(:goal (and (spacecraft_at red s3 s3)))
)
```

Output interpretation

The output of the planner is a sequence of actions that solves the given puzzles. Actions are interpreted as following:

```
move_dir sc_moves sc_static r-c from_c-r to_c-r static_c-r
```

Where `move_dir` indicates if the moving spacecraft goes to the right, left, down or up. `sc_moves` and `sc_static` specify which spacecraft is moving next to the static object. `r-c` says in which row or column are the spacecraft located. `from_c-r` and `to_c-r` are the initial and the final column or row positions of the moving spacecraft and `static_c-r` is the column or row where the static spacecraft is.

In conclusion, the output is interpreted as a list of actions that will always end with a last movement where the red spacecraft moves at the s3 s3 board game position. For the puzzle 2 one possible solution for the final planner output is the following:

```
move_down purple orange s2 s2 s4 s5
move_left green purple s4 s4 s3 s2
move_right red yellow s1 s1 s3 s4
move_down red green s3 s1 s3 s4
```