MAI-PAR: Lunar Lockout Game

Víctor Giménez Ábalos

October 2019

1 Problem definition

A game is defined by a board state. A board is a matrix of (N,M) dimensions (in all cases shown, N=M=5). On the board, there is a goal position G=(G_1,G_2) (in all cases shown, $G_1 = G_2 = 3$). In addition, on the board there is a series of spacecrafts (SP), which occupy a single tile of the board. One of these spacecrafts is red, we will denote it by $SP0$.

The goal of the game is to have the $SP0$ spacecraft standing still at the goal position G. In order to do so, we can move the spacecrafts, but always with a series of concerns in mind:

- A spacecraft can move in an horizontal or vertical direction (so 4 directions).
- A spacecraft must stop its movement if it would collide with another spacecraft.
- A spacecraft can only stop its movement under the aforementioned condition.
- A spacecraft moving off the board is not a valid move.

2 PDDL Representations

In order to implement this in PDDL, we think of two very different representations of the problem, both with pros and cons.

2.1 Dynamic representation

A first representation ($DYNAMIC$) is based on the idea that a whole 'valid movement' can be actually decomposed in several smaller non-valid movements. An example shows it better: if I have a spacecraft in a position P and I want to move it up. Let us say the first obstacle is two rows above (only one empty space between the two spacecrafts). Then, by this representation, the action decomposition goes:

Type	Definition
spacecraft	A spacecraft on the board
position	A position, containing both coordinates x and y

Table 1: Types

Predicate	Parameters	Definition
just_left_of	p1 - *position* p2 - *position*	p1 just left of p2.
just_above_of	p1 - *position* p2 - *position*	p1 just above of p2.
at	sp - *spacecraft* p - *position*	Sp is at position p
movingup	sp - *spacecraft*	We are currently moving sp up.
movingdown	sp - *spacecraft*	We are currently moving sp down.
movingleft	sp - *spacecraft*	We are currently moving sp left.
movingright	sp - *spacecraft*	We are currently moving sp right.
static	-	We are not doing any move.
empty	p - *position*	p has no spacecraft on it.

Table 2: Predicates

1. Start going up.
2. Move up.
3. Next row is not empty, so stop going up.

This is a very simple way of encoding it. Now that the idea is clear, we move to the predicates and actions themselves.

First the types. We have two types, one for denoting a spacecraft, and one for denoting a position. A position represents both coordinates in the board. See Tab. 1

We have several predicates, but most of them are symmetric. We have two predicates for determining position relationship. *just_left_of* a b: indicates that a is neighbour of b, to the left. Symmetrically, *just_above_of* a b: indicates that a is neighbour of b, and above it. We have then a predicate *at* sp p, denoting that the spacecraft sp is at the position p. Finally, and given that our approach is 'dynamic' we need to keep track of what we are doing, and when we can stop doing. The predicate *static* is active only when nothing is moving. Then, we have 4 predicates indicating that a spacecraft is moving in a direction, and has not yet stopped (*movingX* sp, where X is up, down, left or right). Finally we also have the *empty* p, which is a way to denote that a position is empty. It could be check by checking no spacecrafts are on that position, but it would add unnecessary complexity, and a need to check more things. See Tab. 2

In order to represent the actions, we will need 3 of them for each direction. Since they are symmetric, I will explain a generic case, and not entering into much detail regarding checking board positions:

- **startmove**: Starts the movement for a spacecraft. To do so, only precondition is that there is nothing moving: aka. the static predicate is true. As a result, we will now be moving the spacecraft in that specific direction, so we are not static, and we will set the *movingX* sp to true.

- **move**: moves a spacecraft in the direction it was going. We need *movingX* sp, and we will also need that the position to which it would move is empty. As a result, the position the spacecraft was in will be empty, and the position it moves to will no longer be empty (and the *at* of the spacecraft will also change).
- **stopmove**: Stops the move for a moving spacecraft. To do so, it requires that the position we would move to via a move action is not empty. As stated above, this means we have reached another spacecraft or the edge of the board: both valid cases. As a result, *movingX* is no longer true, and we are *static*.

The goal of the problem, however, is not as straightforward as having the spacecraft at the goal position. Rather, we also need the board to be *static* to accept it as a valid solution.

We note that, with this representation, the solver does not optimise the number of valid moves (as expected). It will try to minimise the number of actions performed, which is not what we want. As a fix, we add the fluents library, and consider the cost of a plan the number of 'startmoves' we perform.

Axioms of this representation:

- Always one and only of the *movingX* or the *static* predicates are true at once.
- A position is not *empty* if and only if there is a spacecraft on it (*at*).
- There can never be two spacecrafts *at* a position.

As for the number of possible states, we will not compute them raw, but start by assuming that the problem files will be 'well-formulated', or in concordance with the axioms of the domain.

Assume an (NxM) board, with S spacecrafts. A simple calculation, disregarding the third axiom of the representations, says that the S spacecrafts can be at any position of the board, so $(M * N)^S$, but in addition, for each of these states, we can be moving any spacecraft in any direction (*movingX*), or be static. The final result, thus, would be $(M * N^S * (4 * S + 1))$. For 5 spacecrafts, this is 205078125. However, this answer is wrong since it disregards the fact that we cannot have two spacecrafts at the same position.

For the static part of the algorithm, and knowing each spacecraft can only be in one and only one position, this is a ordered sample without replacement (permutation of smaller size), where the n is the number of positions $(N * M)$ and the k is the number of spacecrafts (S): $\frac{(N*M)!}{((N*M)-S)!}$. However, in each of these possible states, we can be moving a spacecraft $(4 * S)$ or be static (as mentioned above). The resulting possible states are: $(4 * S + 1) * \frac{(N*M)!}{((N*M)-S)!}$.

Type	Definition
spacecraft	A spacecraft on the board
gridindex	A coordinate, containing a single coordinate

Table 3: Types

Predicate	Parameters	Definition
less	i - *gridindex* j - *gridindex*	i's value is less than j's.
just_less	i - *gridindex* j - *gridindex*	i's value is one less than j's.
at	sp - *spacecraft* r - *gridindex* c - *gridindex*	sp is at position (r,c).
empty	r - *gridindex* c - *gridindex*	p has no spacecraft on it.

Table 4: Predicates

For the simple case of a 5x5 board with 5 spacecrafts, we have a total of $(4*5+1)*\frac{25!}{(25-5)!} = 21*6375600 = 133887600$. Do note that most of these are symmetrical since we can permute the helper spacecraft (non-goal) and have the exact same state with respect to the goal (but not with respect to the current 'true' statements).

2.2 Static representation

A second representation (*STATIC*) consists on having exactly four distinct actions, one for moving in each direction. Moreover, these actions always leave the game in a state that is valid as per the rules of the game (and thus its name: static). In addition, we will positions in a less straightforward way, but that allows us to drastically reduce the number of predicates: by single-number coordinates.

First the types. We have two types, one for denoting a spacecraft, and one for denoting an index. A position in the board is represented by two grid indices. See Tab. 3.

We have many less predicates than before. For working with indices, we define two different predicates: *less* i j, and *just_less* i j. The first one represents that index i is less than index j, or seen from coordinates: more to the top or left of the board (depending on whether the indices are for columns or rows). The other one is exactly the same, but it is only for indices that are next to each other. We also have the *empty* i j, mentioned in the section header. We have then a predicate *at* sp i j, denoting that the spacecraft sp is at the position (i,j), where i is the row and j the column, and (1,1) is the top left corner. See Tab. 4.

Each action is a movement of a spacecraft in a direction. For the movement to be valid, we need to see the conditions mentioned in the initial case. Since the 4 actions are symmetric, we only detail the conditions for the *moveup* action, and the detailing on how the predicates change is detailed in the domain file. Generally, though, the things that change are the order of parameters of the less actions, or the row-col behaviour of the *gridindex* parameters.

- The target position is in the same column than the spacecraft.
- The target position row is less than the row of the spacecraft.
- The target position needs to be empty
- The position of the same column but *just_less* than the target position needs to be non-empty (so the one above).
- All positions of the same column whose row is less than the spacecraft but target position row is less than them need to be empty.

The last part of the predicate forces us to check that all of those positions (an array of arbitrary size from 1 to 3) is empty. For that, we will need to use ADL.

The goal this time is just to have the spacecraft at the position.

The axioms of the representation are similar to the ones of static:

- Every action leaves the board after a valid movement by the
- A position is not *empty* if and only if there is a spacecraft on it (*at*).
- There can never be two spacecrafts *at* a position.

As for the number of possible states, it is the same as before, just removing the dynamic part $(4*S+1)$. The final result is $\frac{(N*M)!}{((N*M)-S)!}$, and for our particular case: 6375600. It is much less than before, as expected.

3 Comparison between representations

We see that the second representation, albeit it needs to use ADL, has many less possible states. Reducing the search space by a magnitude of 21 is quite good.

We note, however, that even though the first option has a big state-space, the branching of most of its actions is deterministic: when performing the 'start-move' action, the consequent actions that will come after is not decided by the planner, but is already pre-established by the state of the board when we began moving. Thus, the search is not necessarily more 'complex' than the one performed by the static method.

However, in any case, the static method seems to be better in general. The advantages are: less actions to explore, less predicates, less state-space possibilities, no need to use fluents... The only arguable disadvantage is that we require ADL, which is not available for all planners.

4 Generating problems, running and understanding output

For generating a new problem for the domain, a python script ‘problem_header_generator.py‘ has been provided. This simple file contains two problem generator classes, one for static and one for dynamic, as well as three basic problem definitions.

To create a new problem, one needs to create a dictionary containing the board dimensions, the spacecraft’s positions (top left is (1,1)) and with the red spacecraft as the first one, and finally the location of the goal (same format as spacecrafts). There are 4 examples of them in the file. Set the variable ‘simple_problem‘ to your dictionary.

To specify the format (static-dynamic), change the class instantiation ‘problem = DomainX‘. X=1 means dynamic domain, X=2 means static domain. Finally, run ‘python problem_header_generator.py ¿ out.pddl‘, to generate the problem file. On top, it will display an ascii image on how the board looks, to ensure the problem generated is as you define.

Disclaimer: the static domain has a limitation coming from the fact that we use gridindices, and should not be used with board dimmensions that are not square.

For running the planner, I recommend Metric-FF. Download and compile it. Run it as you would normally run Metric-FF, with flags -o as the domain you want to use and -f with the problem. In experiments, there has been times where the optimal solution was not found directly, or it stated that such a solution did not exist: if such a thing happens, it is encouraged to change the -s flag. In the case of static domain, try flag values: {0,1,2}. In dynamic, {3,4,5}.

Interpreting the output is different in each domain, obviously.

Dynamic: one can understand the plan just by looking at the *startmoveX* and the name of the spacecraft. Where each spacecraft is at the beginning can be seen at the top of the problem file if it was generated by the script. The actions in between can be used to count the number of spaces it moves in each movement if one wants to, but it is unnecessary.

Static: each action decomposition can be understood as the following: the first three parameters are the spacecraft and its current location. The fourth parameter is the ’destination’, or the coordinate it moves to. If it is a vertical movement, it displays the row where the movement ends.