Moving Target Problem

Josep de Cid

November 2019

1 Introduction

The Moving Target problem, consists in a $N \times N$ board game with two players, that we will identify as the Robot (**R**) and the Ghost (**G**). The goal is that, "playing" as R and knowing the trajectory that G will follow *a priori*, to come up with a plan to reach G before it finishes his path and trying to do it with the minimum possible cost.

Each cell can have a cost to pass through, which is used to compute the total cost. R can move to the adjacent cells in the 4-neighbourhood and stay in the same cell, having a total of 5 actions. Staying in the same cell, adds the score of that cell again.

2 Domain Representation

The representation presented, is the implemented in the attached `pddl` file: `moving_target_domain.pddl`. Its summary can be seen in table 1.

Objects	Predicates (States)	Actions	Functions
R, G	at		cost
C0, C1, ..., C_{N-1}	next	move	
T0, T1, ..., T_{T-1}	scheduled		path-cost

Table 1: Summary of Objects, Predicates, Actions and Functions

Objects

There are two different objects in our problem, the `player` and the `value` (inheriting from `object`), divided in `robot` and `ghost` and `coord` and `time` respectively. In the domain file we define these types:

```
(:types value - object
        coord time - value
        player - object
        robot ghost - player)
```

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

```
(:objects R - robot
          G - ghost
          C0 C1 C2 C3 - coord
          T0 T1 T2 T3 - time)
```

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 $1 < 2 < \ldots < 5$, defining if two indices are adjacent.

Predicates

We define 3 different predicates, two for position checking and one for movement purposes:

```
(at ?p - player ?x ?y - coord ?t - time)
```

The player p is at (x, y) at timestep t. Only one predicate at a time for the same player is true. Past timesteps predicates are set as false.

```
(scheduled ?x ?y - coord ?t - time)
```

Defines the scheduled path of the ghost, that at t is located at (x, y). These are initialized at the beginning and aren't modified during the execution.

```
(next ?a ?b - value)
```

Indicates that the value b is immediately after a. It's used in `time` instances to indicate that timestep t is followed by timestep $t + 1$ and in `coord` instances to indicate that c is next to $c + 1$ and vice versa (bidirectional representation).

Functions, Metrics and Goals

We propose two functions, `path-cost`, which counts the cost to reach the plan goal, get in the same (x, y) and t the Robot and the Ghost, adding up the costs of the cells that the robot has walked through. It is initialized as 0 with `(= path-cost 0)` in the problem files. The second function indicates the cost of each cell (x, y), defined as

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

and `path-cost` increased by the corresponding cell value as an effect of any action with `(increase (movements) (cost ?x ?y))`.

These metrics are used later in the problem metrics to `minimize` the cost required to complete the problem and obtain the cheapest solution:

```
(:goal (exists (?x ?y - coord ?t - time)
               (and (at R ?x ?y ?t) (at G ?x ?y ?t))))
(:metric minimize (path-cost))
```

Actions

The domain has 5 actions but for the implementation we merged them in 1, to simplify, called `move`.

Parameters

The function receives twelve parameters, the robot, it's current coordinates and next step coordinates, the same for the ghost and the current and future timesteps.

```
:parameters (?r - robot ?xr ?yr ?xrn ?yrn - coord
             ?g - ghost ?xg ?yg ?xgn ?ygn - coord
             ?t ?tn - time)
```

Precondition

In the precondition we must start checking some basic conditions to be satisfied, such as than the robot and ghost are at the given positions in the current timestep, that the timesteps are consecutive, and the ghost is schedulet to go to the next specified position.

```
(at ?r ?xr ?yr ?t)           ; Robot at (xr, yr) at t
(at ?g ?xg ?yg ?t)           ; Ghost at (xg, yg) at t
(next ?t ?tn)                ; tn = t + 1
(scheduled ?xgn ?ygn ?tn)    ; Ghost at (xgn, ygn) at tn
```

We also must check that there next cells are the same cell, or the adjacents in the 4-neighbourhood.

```
(or                                    ; Next robot position:
   (and (= ?xr ?xrn) (next ?yr ?yrn)) ; - Adjacent on L or R
   (and (next ?xr ?xrn) (= ?yr ?yrn)) ; - Adjacent on T or B
   (and (= ?xr ?xrn) (= ?yr ?yrn))    ; - Remain same cell
)
```

Effects

The main effect is that the robot is not anymore at the original position and now it is at the next one, the same for the ghost. As commented in the functions section, we increase the number of movements by one:

```
:effect (and
    (not (at ?r ?xr ?yr ?t)) ; Robot not anymore at (xr, yr)
    (at ?r ?xrn ?yrn ?tn)    ; Robot is now at (xrn, yrn)

    (not (at ?g ?xg ?yg ?t)) ; Ghost not anymore at (xg, yg)
    (at ?g ?xgn ?ygn ?tn)    ; Ghost is now at (xgn, ygn)

    (increase (path-cost) (cost ?xrn ?yrn))
)
```

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 2 players and N^2 cells, the predicate `at` can create $2 \cdot N^2$ different states. The predicate `scheduled` adds $\mathcal{T}$ more to the count, and `next` has $\mathcal{T} - 1$ predicates for the time and $2 \cdot (N - 1)$ for the cells. However, this number is so easy to cut down. Considering the ghost path as already defined, we can treat it as a constant. Hence, we have $\mathcal{O}(N^2 \cdot \mathcal{T})$ possible states.

3 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 moving_target_domain.pddl -f moving_target_problem_X.pddl
```

where `X` is the number of the problem. 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 R C0 C0 C1 C0 G C3 C3 C2 C3 T0 T1)
(move R C1 C0 C1 C1 G C2 C3 C1 C3 T1 T2)
(move R C1 C1 C1 C2 G C1 C3 C1 C2 T2 T3)
```

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.

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

To create a new problem it's only necessary to adjust value of the variables `problem_number` and `config` dict, that contain the board size, initial positions, cost matrix and ghost path.

```
problem_number = 1
config = {
    'N': 4,
    'R': (0, 0),
    'G': (3, 3),
    'sequence': [(3, 3), (2, 3), (1, 3), (1, 2)]
}
```

The script creates the specific *init* section of the problem, with all the predicates, 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 a problem is attached in `plan.txt`, use the `--file` option in the script. We also provide a few already codified problems to test, the example in figure 1, an unsolvable case and a more difficult one.

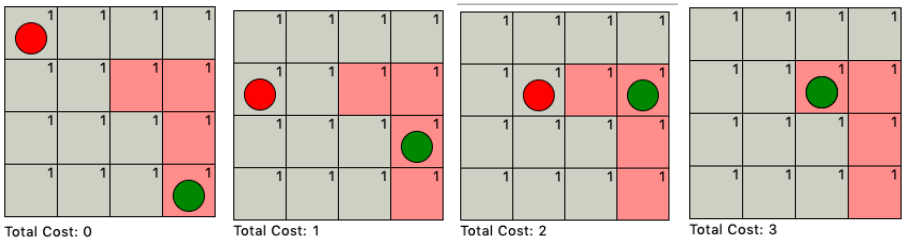

Figure 1: Visualization for problem 1

4 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, even in this case, that we are modeling time sequences.

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