Homework 2 - PAR-MAI

Jordi Armengol

November 2019

1. In this assignment, you will be writing the required PDDL files for a planner to help the robot catching the moving target. The task of the planners is to generate the path of the robot should follow in order to catch the target. You could suggest a planner in x and y domain, but if you can implement it with x, y and t (time-step) domain, there a bonus degree for that

   Answer:

Introduction

The solution is partially inspired by the domain of the previous problem (Lunar Knockout), with some key differences. The general idea is that in the domain we are just going to let the robot either move in one of the four directions or stay in the same cell, while in the problem we are going to implicitly specify the movement of the target by marking its trajectory (position and time) as the desired ending state.

Initial interpretation and considerations

Interpretation of the movement of the target

If we consider that the target disappears once it has arrived to its final solution, the example provided by the assignment is not solvable. If we do, it is. Our domain is agnostic to this interpretation, it works fine with both of them. Our problem instances are not, but accepting the second interpretation is as easy as repeating the final position of the target with different time steps.

Considerations on the need of using the fluents extension

If we assume that the cost of each cell is 1 and we use the FF planner, the `fluents` extension is not really needed, because in the case that no metric is specified, FF optimizes the plan length, which is equivalent to the number of time steps, which in turn is equivalent to the cost of the solution. However, we have still explicitly encoded the cost and its optimization with `fluents`, for the sake of clarity and portability to other planners with different defaults. However, if we comment out all the `fluents` elements from the solution, the code will work exactly equal for the reason stated above.

Considerations on the maximum cost

The restriction on the maximum cost is implicit in the sense that time steps that are not introduced in the problem instances are just not considered.

A note on the board coordinates

In the first exercise (Lunar Knockout), the board coordinates were not provided by the example, and I interpreted that the first coordinate corresponded to the X axis and the second one corresponded to the Y axis. In this case, since the board coordinates are provided in the assignment, and the first coordinate is the one that indicates the vertical axis, we will be denoting coordinates with Y, X.

Notice that for us, going up is going from (1,0) to (0,0), for instance, while going down means going from (2,1) to (3,1). So, going up means decreasing the Y coordinate while keeping constant the X coordinate.

Solution

Specifically, this solution is composed of the following elements:

- **Types**: Initially, `Coord` (Coordinate) and `Time` for the spatial and temporal coordinates, respectively. However, in order to have more compact problem files, they can be further unified into a single type, `Coord`, and consider time just as a third coordinate. Thus, we do not have to manually implement arithmetic for both types. It could be the case that the planner had to check for additional, unnecessary possibilities (eg. if our problem requires more time steps than spatial coordinates, the planner will have to check the pre-conditions for spatial coordinates bigger than the actual ones), but it is a trade-off and in this case the legibility of the solution is more important.
  Notice that **we are suggesting a planner with x, y and t (timestep) domain**.
  Note that with the `fluents` it would be possible to avoid defining coordinates as objects and their corresponding order relations. Nevertheless, since `fluents` is not strictly required for solving this problem (as we will see later on) and for the sake of clarity and learning *pure* PDDL, we will leave the solution with manually defined coordinates.

- **Objects:**
  - `Coord`: One for each coordinate and time step. In the case of the example provided in the assignment, since we have a 3x3 grid, we will have `zero`, `one`, `two`, and `three`. In order to make the example problem solvable, we will need two additional instances, `four`, and `five`.
- **States (predicates and variables/functions):** Intuitively, the states are composed of the position of the robot, the position of the target, the current time step and the cumulative cost of the solution. In the particular case of this assignment, since all cells have cost = 1, the cost and the current time step can be thought as equivalent, but we are noit going to unify them in our solution in order to make it potentially more general.
  In terms of predicates (which in PDDL compose the states, together with the variables/functions if we use the `fluents` extension), the relevant predicate is the one that states the spatio-temporal position of the robot, `locatedAt coordX coordY time`. The other predicate of the solution, `exactlyOneLessThan ?a - Coord ?b - Coord`, which means $a + 1 = b$, is constant, since we are not going to dynamically modify the arithmetical order relationship of our coordinates and time steps.
  Notice that it is not possible to explicitly encode the position of the robot in our domain. Instead, it is implicitly encoded in the problem by stating the positions and time steps of the target as a conjunction of goal states. Since the trajectory of the target is deterministic and known in advance, its trajectory does not affect the number of possible states of a given problem, which will actually be the number of combinations of the feasible spatio-temporal coordinates of the robot ($N \times N \times \#$`time_steps`).
- **Actions:** (go) up, down, left, and right (the four possible movements of the robot), and stay (ie. no movement in that time step). The five actions are quite similar:
  - Pre-condition: We check that the next cell is effectively next to the current one and that the next time step is, arithmetically, the next one.
  - Effect:
    - Delete list: The previous position (except in the case of the neutral movement) and time step do not hold anymore.
    - Add list: One of the two spatial coordinates is modified and the time step is increased. In addition, if we use `fluents`, the cost is increased (by 1, as stated in the assignment).
- **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:

  - The robot is at one and only one position for a given time step.
  - The robot can only move up, down, left, right within the board and towards a cell that is immediately next to the previous one, or stay at the same position.
  - The movement per each time step is limited to one cell.
  - Vising each cell has a cost of 1.

- **Problem instances**: All problem specifications require to declare the aforementioned objects and the exactly less than one relationships, since PDDL has no inherent notion of arithmetic (unless we use the fluents extension, which we do not for the spatio-temporal coordinates). We must declare the initial position of the robot, as well as initialize the cost, which we want to minimize, to 0.
  With regard to the target, its trajectory is encoded in the goal. We define a logical disjunction for its trajectory. For instance, in the case of the example provided in the assignment, we state that the robot should be located at position (3,3) in the initial time step, or at position (2,3) in the second time step, and so on. If we interpret that when reaching the final position the target does not disappear, we just repeat the goal for some time steps (eg. either located at position (2,2) in the third time step, or located at position (2,2) in the fourth time step, or...).

Implementation

As attached files, we have a domain file, `domain.pddl`, and two problem files, `problem_ex.pddl` and `problem_rand.pddl`, corresponding to the example in the assignment and a random instance generated by our problem generator. Notice that this script for automatically generating problems has been included in the delivery as well, in *problem_gen.py*.

For generating plans, just run a planner such as FF (the Metric version if the `fluents` elements are not commented out) or an online service such as this one specifying the respective domain and problem files.

The output of the first problem with FF-Metric is as follows (some parts of the output have been removed for the sake of simplicity):

```
ff: found legal plan as follows

step    0: RIGHT ZERO ZERO ONE ZERO ONE
        1: RIGHT ONE ZERO TWO ONE TWO
        2: DOWN TWO ZERO ONE TWO THREE
        3: DOWN TWO ONE TWO THREE FOUR
        4: REACH-GOAL
```

```
time spent:    0.00 seconds instantiating 780 easy...
               ...
               0.00 seconds total time
```

The full 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, `0:  RIGHT ZERO ZERO ONE ZERO ONE` means that the robot must go right in the first step. The rest are just the parameters that were used in the action (in this case: the robot is at (0,0), it will move to (zero, one) and the time step will increase from 0 to 1). So, our planner suggests:

(a) Move robot right.

(b) Move robot right.

(c) Move robot down.

(d) Move robot down.

which is a correct solution to the first example, and an optimal one, assuming that the target remains in the final position once it has arrived. Otherwise, the problem is not solvable.

In the case of the randomly generated instance, with:

```
 python3.6 problem_gen.py problem_rand random 10 6
    > problem_rand.pddl
```

we generate the following problem:

```
...
    (locatedAt zero three zero)

    (= (cost) 0)
)

(:goal (or
    (locatedAt three zero zero)
    (locatedAt three zero zero)
    (locatedAt two zero one)
    (locatedAt two one two)
    (locatedAt two two three)
    (locatedAt two two four)
    (locatedAt one two five)
))
...
```

So, the robot starts at (0,3). The target will follow this trajectory: (3,0), (3,0), (2,0), (2,1), (2,2), (2,2), (1,2).

The generated plan is the following:

```
ff: found legal plan as follows

step    0: DOWN THREE ZERO ONE ZERO ONE
        1: DOWN THREE ONE TWO ONE TWO
        2: LEFT THREE TWO TWO TWO THREE
        3: REACH-GOAL


time spent:    0.00 seconds instantiating 1302 easy...
               ...
               0.00 seconds total time
```

which means:

(a) Move robot down.

(b) Move robot down.

(c) Move robot left.

which is a correct solution to the randomly generated example, and an optimal one.

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

```
python3.6 <problem_name> robot <x> <y> target <x> <y> [x1, y1]
    > <problem_name>.pddl
```

with x, y in [0-3] and such that [x1, y1] is a valid trajectory for the target. For instance:

```
python3.6 problem_gen.py problem_ex robot 0 0 target 3 3 2 3 2 2
    > problem_ex.pddl
```

Generates the problem for the example provided in the assignment (assuming that the target disappears once it arrives to the target; otherwise the user can repeat the final position (2 2) three times).

The script can generate random instances as well:

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

For instance:

```
 python3.6 problem_gen.py problem_rand random 0 4
    > problem_rand.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 the coordinates of the trajectory of the target are consecutive), but it is not guaranteed that the problem will have a solution, unless the final position is repeated.