Planning with numbers and optimization

Laboratory Practice 1 – Part 2/3 – Automatic Planning – Academic Year 2023-24

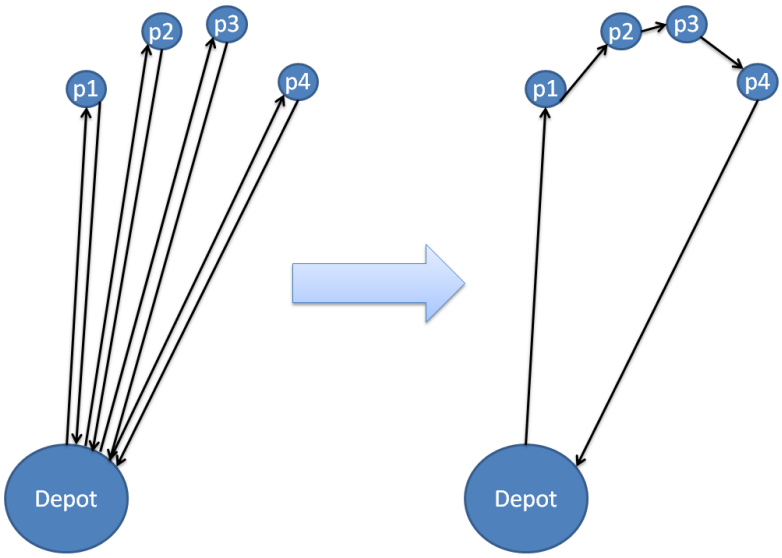
Memorandum of Practice

For each of the three parts of practice 1 it will be necessary to write a short report discussing the results of each exercise. There is no minimum or maximum number of pages for the report, it must simply explain in a clear and understandable way the results that are requested. Some of the planners' output can be included, if it helps explain the results, as well as tables when they can help synthesize and compare the results of different tests. The report must be submitted in PDF. In each exercise, it will be indicated what must be explained in the report.

Exercise 2.1: Emergency Service, Transporters

Until now, a drone could carry a maximum of two boxes, one per arm. The planner considered which arm he carried each box on. Expanding the number of arms of the planner to improve its load capacity is inefficient, since the planner considers different states for each possible combination in which each of the arms gets each of the boxes, which causes the search space (the number of possible states) to grow exponentially with the number of arms without this resulting in any benefit for the resolution of the problem.

In this exercise we will consider an alternative way of moving boxes using conveyors, structures that will allow the drone to fly carrying several boxes at the same time, but where order is not relevant. The conveyor would be like a container where several boxes can fit, which must be put in or taken out of it with explicit actions. A drone can load up to four boxes onto a conveyor, which must be in the same location. It can be assumed that the conveyors are initially at the warehouse location, as are all the boxes. Once the boxes are loaded, you can take the conveyor to a location where there are people in need of supplies and unload one or more boxes from it. You can continue to take the conveyor to another location, unload additional boxes, etc., and you don't have to return to the depot until all the boxes on the conveyor have been delivered.



Although currently only a single transporter is required for the single drone used in the problems, a specific type must be created for the transporter. It is necessary to know what and

how many boxes are on a conveyor, to avoid loading it beyond its capacity. The capacity of the protractors will be the same for all of them and can be specified in each problem, so it will be part of the problem file and not the domain.

It is important not to introduce specific grooves in the conveyors (e.g., slot1, slot2, slot3, etc.) as this would increase the problem state space unnecessarily, causing redundant states to exist, as would happen if we increased the number of arms that the drone has. For example, if we were to use slots, we would consider two different states, although completely equivalent from the logic of the problem, if box A is in slot 1 and box B in slot 2 than if box A is in slot 2 and box B is in slot 1. Redundant states would cause the search space to expand unnecessarily, and lead to more processing time on the part of the planner when they must solve a problem. For this reason, we should not record the position of the box, but simply which boxes there are and how many there are in total.

Numbers by Predicates

Since many planners do not support numeric values, there is always the possibility of representing numbers using standard PDDL objects and predicates. To do this, we will first declare a type of numeric data:

```
(:types ... num ...)
```

Then we'll declare objects that represent different integers:

```
(:objects ... N0 N1 N2 N3 N4 N5 N6 N7 – num)
```

For the planner these are simply objects, so we will have to define any operation we want to perform with them using predicates:

```
(:predicates (next ?numA ?numB – num)
```

In the initialization of the problem, it will be necessary to establish the relationships between numbers:

```
(:init ... (Next N0 N1) (next n1 n2) ... (Next N6 N7)
```

Then we can use the objects in the preconditions and effects of the actions. For example, in the following action in a planning domain with elevators, the action of moving up a floor is implemented as follows:

```
(:action up-floor
      :parameters (?a – elevator ?from ?to – num)
      :precondition (and
            (in ?elevator ?from)
            (next ?from ?to))
      :effects (and
            (not (in ?elevator ?from))
            (in ?elevator ?to)))
```

Not being defined `(next n7 ...)`, when the elevator is on the 7th floor, the precondition `(next ?from ?to)` can never be met, so it will not be able to go any further. On the other hand, it is not necessary to define the predicate "previous" as opposed to "next", since the latter can also be used the other way around, to verify whether one number is prior to another.

Actions

Depending on how the domain was structured for the previous exercise, it is likely that it will be necessary to define at least three new operators for this exercise: put-box-on-conveyor, move-conveyor, and pick-box-from-conveyor. The action of putting a box on the conveyor should be preceded by an action of picking up the box by the drone. Likewise, the action of removing a box from the carrier will continue with the action of delivering that box to someone else.

In this exercise, you should do the following:

1. Modify the domain as explained above. This new domain must be made in a new directory to keep the previous versions of the domain intact.
2. Adapt the problem builder to generate problems according to the new domain, including initialization of numeric values and protractors.
3. Verifies that the generated domains and issues can be solved correctly using the new actions related to the conveyor. Test the planners in Part 1 on small-sized problems. Which planners make use of conveyors? Which ones don't?

Exercise 2.2: Emergency Service, Action Costs

As stated above, the intention behind the use of conveyors is that a drone does not need to return to the depot for each box. However, it is not obvious to a planner that going back and forth long distances is inefficient, since we are not telling them the cost of the trip. In fact, using the conveyor can result in longer plans, at least if we consider only the number of shares. Therefore, it is necessary to add costs to the different actions in the domain.

Action Costs

Using the knowledge seen in the laboratory, he expands the domain and previous problems by introducing action costs to ensure that flying between great distances has a comparatively high cost with picking up and dropping boxes in the same location. To do this, modify the problem generator and make use of the `flight_cost()` function, which specifies the cost of flying between randomly generated locations.

To add costs to actions, you must do the following:

1. Add :action-costs to the requirements section of the domain specification. This requirement limits the optimization functionality in PDDL to the minimization of a so-called "total-cost" flow, which represents the cost of actions.
2. Define a total cost function called "total-cost" and a return cost function "fly-cost" that receives the origin and destination of the flight as a parameter and returns its cost.
3. Increase the total cost by a fixed amount as an effect of those actions with constant cost.
4. In flight actions, add an effect that increases the total cost based on the return cost between origin and destination.
5. Modify the problem builder so that it starts the total cost to 0.
6. Modify the troublemaker to initialize the values of the fly-cost function between each pair of locations. You must use the flight_cost() function to do this. Costs will always be integers, not negatives.
7. Add a total cost minimization metric to the troubleshooter.

8. Use the generator to generate some test problems of different sizes, always keeping 1 drone and 1 transporter with size 4.
9. Find out what size problem the following schedulers can solve in a maximum time of 1 minute to return a first solution: Metric-FF and Fastdownard with the aliases LAMA (lama), BJOLP (seq-opt-bjolp), and FDSS2 (seq-opt-fdss2). Create a table indicating for each planner the size of the problem solved, the cost of the solution provided and whether the planners are optimal, *anytime* or *portfolio.*