Assignment report for the course Automated Planning: Theory and Practice

Master's degree in Artificial Intelligence Systems; A.Y 2022/2023; Marco Roveri

Joy Battocchio Davide Guidolin

I. Introduction

The following report aims to give a detailed explanation of the solutions of the 5 problems proposed in the assignment. The problems considered a scenario where one or more robots have to deliver emergency supplies to injured people located in space. These were the initial explicit assumptions:

- Each injured person is at a specific location, and does not move.
- Each box is initially at a specific location and can be filled with a specific content if empty. Box contents shall be modeled in a generic way, so that new contents can easily be introduced in the problem instance.
- Each person either has or does not have a box with a specific content.
- There can be more than one person at any given location.
- The robotic agents can move directly between arbitrary locations.
- We want to be able to expand this domain for multiple robotic agents in the future.

While the suggested actions were the following:

- Fill a box with a content. Robot box and content must be in the same place, box must be empty.
- Empty a box, satisfying the person at that specific location.
- Pick up a box located in the same place as the robot.
- Move to another location.
- Deliver a box.

Based on these assumptions here is the formulated solution used in all problems. We considered 5 main types:

- Robot
- Person
- Location
- Box
- Content

and 5 main actions:

- Fill
- Empty
- Load
- Move
- Unload

The main workflow the robot follow to satify a person is to **fill** a box with a content, **load** the box on itself and **move** carrying the box toward the destination. Once it gets where the person is located it **unloads** the box and **empties** it giving the content to the person. After that the robot needs to restart in order to satisfy another person, so it can re-use the same box as before and carry it to the place where the supplies are stored, or look for another box. We assume the supplies are infinite where stored, but only one portion can fit in a box, so if 2 people need the same supply the robot must re-fill with it.

For doing so the folllowing informations are needed:

- Where a robot is (**robot-at**).
- If a robot is carrying a box(**robot-loaded**).
- Which box a robot is carrying (**robot-has**).
- If a box is full (**box-full**).
- Where a box is (**box-at**).
- What is the content of a box (**box-contains**).
- Where a supply is stored (**content-at**); specified in the initial assumptions of the problem and fixed.
- Where a person is (**person-at**); specified in the initial assumptions of the problem and fixed.
- What supply a person has (**person-has**); which represent the goal.

For all problems we used **negative preconditions** since the predicates make more sense this way and it is supported by almost every planner. We also made a version without **negative precondition** that has very few differences, it is sufficient to use **box-empty** and **robot-unloaded** and remove the negated preconditions from the **move** action. Notice that this configuration ensure there are no constraints on the number of robots or boxes in order to satisfy any number of people.

II. Problem 1

The first problem was solved simply by implementing what is described in the introduction above. We set the environment with 1 robot, 3 people, 5 locations (one of which is the depot where the supplies are stored), 2 boxes and 3 types of supply. Initially the 3 people are scattered over the locations, while robot, boxes and supplies are in the depot. The goal is to deliver medicine to person2, and to deliver food and medicine to person3; person1 does not need anything.

III. Problem 2

The second problem required to think about the **move** action in a different persective, ensuring that a robot can load the

boxes on a carrier instead of itself, and that the carrier can be transported by a robot to the destinations.
The solution we developed are two, the first one is simpler since it only has an additional type **carrier** and its **capacity** is represented through a function. The initial capacity of each carrier can be specified in the problem, and it is managed through the usage of fluents: **load** can be performed only if the capacity is greater than 0, and the effect is to decrement it, the opposite for **unload**.
Another solution has been implemented since the first one relies on the **fluents** requirement, which is not supported by many planner we used. We took inspiration by an example provided during the lab session, in which a new type called **capacity-number** is used to described all the possible capacities. Then for each **load** and **unload** operation done on the carrier is important not to forget to specify the precondition **capacity-predecessor** through which we ensure the numbers we use to upload the capacity are consequent. The order of the capacity is specified in the problem as follows:

```
(capacity_predecessor capacity_0 capacity_1)
(capacity_predecessor capacity_1 capacity_2)
(capacity_predecessor capacity_2 capacity_3)
(capacity_predecessor capacity_3 capacity_4)
```

Plus the initial capacity of each carrier (4 as requested):

```
(capacity carrier1 capacity_4)
```

Finally the last modification with respect to the first problem is the **move** action, which no longer allow a robot to carry a box, but a carrier instead. Thanks to this any robot can carry any carrier, even if loaded by another robot.
To test it with a more complex environment we increased the number of people to 6, the boxes to 5, the locations to 8 and the supplies to 5, the robot and the carrier are just one each, as requested in the assignement. Of course there are no constraints neither in the number of robots nor of carriers. From this point on, all next problems are solved using solution 2 described above as base.

IV. Problem 3

Problem number 3 required a hierarchical implementation of the actions. We decided to construct it as described in the diagram VII The main task is to **satisfy** a person delivering to him/her a specific supply. To do that the robot first has to **get the content**, then to **delivery** it. To **get the content** the robot **gets to** the place where the supply is stored, and **loads** it on the carrier (**fill + load**). The **delivery** consists of **get to** the place where the person is and **unload the content** (**unload + empty**). This is a standard workflow, of which there are some variations listed here:

- A robot could already have the needed supply in a box it is carrying. For this we have an alternative method for the **get_content** task, which is **m_already_got_content** that is implemented by the action **no_load**, which basically does nothing.
- A robot could already be where it has to go, so the **get_to** task is also implemented using the **m_already_there** method that calls the **no_move** action.
- In case the number of boxes is less than the number of people to satisfy, A robot must keep the box after delivering the supplies, to do that there are 2 implementations for the **delivery** action, one that **load** the box on the carrier after it empties it and one that does not.
- When a robot wants to re-use a box, it needs to **get_content** using a box it is already in the carrier, for this reason there are 2 implementations of the **get_content** task, one **unload** a box from the carrier before filling it and load it, the other does not.

To test it we kept the same environment as in problem 2, with 7 **satisfy** tasks to perform. We had to specify an ordering to keep the searching space sufficiently small and to find a plan in little time, if no temporal constraints are put, the ordering can be removed.

V. Problem 4

For solving problem 4 we were asked to use durative actions. The chosen duration are the following:

- Fill: 5
- Load: 4
- Move: 10
- Unload: 2
- Empty: 3

We considered the robots to have one arm, this means it cannot load several boxes at the same time, or loading and filling as well as unloading and emptying at the same time.
The only case possible where a robot could make two actions in parallel are if it starts unloading a box from the carrier while still moving. To manage the possibility and not possibility to parallelize actions the predicates **busy** has been added. It refers to a robot and is used to understand when the successive action can begin. In fact all the non-parallel actions have the same effects:

```
and     (at start (busy ?r))
        (at end (not (busy ?r))
        ...
```

and the same precondition:

```
and     (at start (not(busy ?r)))
        ...
```

The actions **empty**, **fill** and **load** have as precondition $(overall(robot - at?r?x))$ meaning that for all the execution of that action the robot must be at that location.
The **unload** action instead has as precondition $(atend(robot - at?r?x))$ which means that we only need to be at the right location when the box touches the ground, so when the action **unload** ends.

VI. Problem 5

For solving problem 5 we were asked to implement problem 4 using PlanSys2 [1] that is a software platform used to

create, execute and monitor plans on a robotic agent. We used PlanSys2 inside the Robot Operating System 2 (ROS2) [2].

To implement the problem in PlanSys2 we started from a simple example [1] and we modified the necessary files that are:

- `launch/launcher.py`
  that contains the code to select the domain and run the executables that implement the PDDL actions. Here we only changed the names of the actions and the project name.
- `CMakeLists.txt` and `package.xml`
  in which we only had to change the project name
- `launch/commands`
  that contains the definition of the problem
- `pddl/domain.pddl`
  that contains the pddl domain
- all the files in `src/`
  that contain the implementation of the actions

Since we didn't have a real robot, we used fake actions that simulate the behaviour of a robot by waiting for an appropriate amount of time before ending the action and by logging what the robot should do.

PlanSys2 supports two planners: TFD [3] and POPF [4]. We tried both the planners because POPF was slow in finding the plan and it was stopped early inside the `plansys2_terminal` while TFD was able to find the plan faster but, since it isn't the default planner in PlanSys2 it required more work for the setup. Moreover POPF does not support the `negative-preconditions` requirement so the domain was slightly different from the one used in TFD.

Regarding the TFD setup we had to modify the source code because PlanSys2 was not able to find the correct TFD path, however everything is explained in the Github repository [2].

VII. Appendix

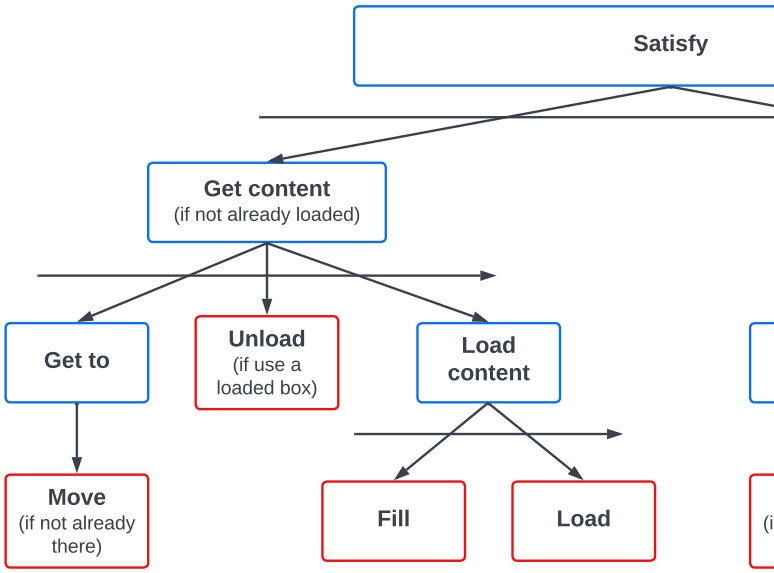
References

[1] F. Martín, J. Ginés, V. Matellán, and F. J. Rodríguez, "Plansys2: A planning system framework for ros2," 2021.

[2] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022.

[3] "Temporal fast downward." http://gki.informatik.uni-freiburg.de/tools/tfd/.

[4] "Forward-chaining partial-order planning." https://nms.kcl.ac.uk/planning/software/popf.html.



[1] https://github.com/PlanSys2/ros2_planning_system_examples/tree/master/plansys2_simple_example

[2] https://github.com/Davide-Guidolin/Automated-Planning-Project