Automated Planning Theory and Practice Project Report

Matteo Farina & Wamiq Raza
University of Trento
MSc in Artificial Intelligence Systems
{matteo.farina-1, wamiq.raza}@studenti.unitn.it

January 15, 2022

1 Introduction

In this report we explain our work on PDDL and PlanSys2, which is structured in four problems each building on the previous one. This introductory section contains details related to both (i) the structure of the delivered zip archive and (ii) the rest of this document.

1.1 Archive content

The zip archive contains 4 main folders, denoted by problem number. In case multiple solutions are provided for a single problem, the related problem folder will contain subfolders having a representative name for the solution they contain. Each folder containing a solution consists of three main elements:

- the **src** directory, which contains the proposed **domain.pddl** and **problem.pddl** files;
- a **command.txt** file, which contains the shell command launched in order to generate a plan with the files in the **src** folder. Analyzing the content of this file will let you know **which planner has been used** to parse the domain-problem pair and generate a plan;
- an **output_plan.txt** file, which contains the output of the command in **command.txt**.

. Note that an exception is represented by the *problem_4* folder, which contains the source code for the PlanSys2 infrastructure within the *plansys2_assignment* subfolder.

1.2 Report content

The content of the report is organized as follows: in Section 2 we discuss a possible solution for problem 3.1; in Section 3 we present two possible solutions for problem 3.2; in Section 4 we discuss a solution for problem 3.3 and show that our implementation successfully supports concurrent actions; finally, in Section 5 we describe a solution to the last problem within the PlanSys2 framework.

2 Problem 3.1 Approach

In the first problem the domain file contains a total of three actions: **move, load** and **unload**. The domain file also contains a constant, i.e. our robotic agent, since the assignment states *"[...] robotic agents, which currently happens to have a single member in all problem instances"*. Figure 1 represents the domain hierarchy of the defined types. Since the assignment wanted us to model crate contents in a generic way, we decided to define them as subtypes of crates, categorizing its content. Thus, in order to add/delete a specific crate content, one simply has to add/remove a type in the domain definition. An alternative way of modeling this is shown in the section dedicated to temporal planning. The LAMA planner has been used to generate a plan for this part of the assignment.

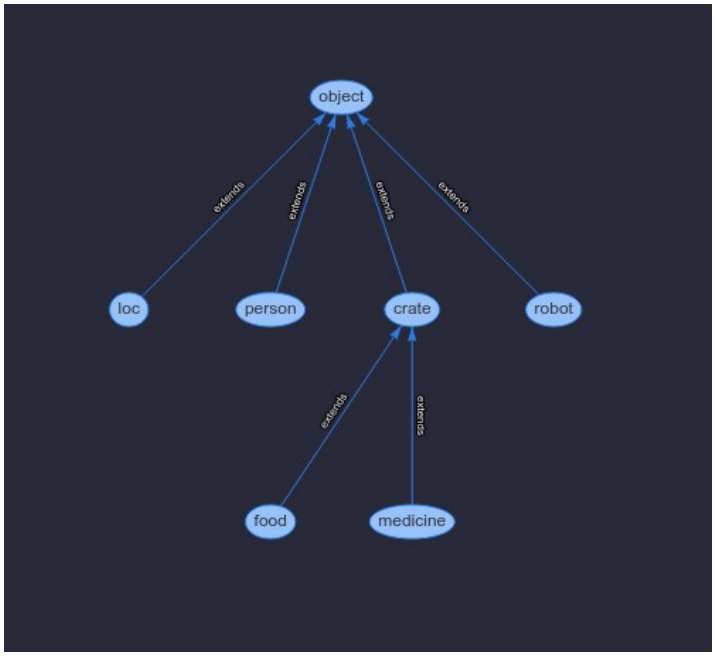

Figure 1: Domain hierarchy for problem 3.1.

2.1 Predicates Description

We have used typical localization predicates (**robot_at**, **person_at** and **crate_at**), which define the exact position of different entities in the space. Note that defining multiple predicates could be avoided by adding, for instance, a **locatable** type as the supertype of **robot**, **person** and **crate**, but we found it more intuitive and comprehensive to have different predicates as a design choice.
The **has_crate** and **carrying** predicates identify whether a crate is held by either a person or a robot, respectively. Together with the **crate_at** predicate, these create the "lifecycle" of a crate, which can either be (i) at a specific location or (ii) held by an entity, be it a person or a robot. The last predicate **free** denotes whether a robot can load a crate or not, being it busy already with some other crate.

2.2 Actions Description

We propose three actions: **move**, **load** and **unload**. The **move** action allows a robot to move from a source location to a destination location. There is no adjacency check since *"the robotic agents can move directly between arbitrary locations"* is stated by the problem description. The **load** action allows a robot to pick up a crate, in case the robot hasn't loaded another crate yet and both are at the same location. The **unload** action allows a robotic agent to deliver the content of a crate it is holding to a given person, being at the same location.

2.3 Goal Description

Since we wanted to have people with different needs, we defined the goal as a conjunction of disjunctions with some people needing only food, others needing only medicines, others needing both and others not needing anything. To understand whether a person is served, we simply check the **has_crate** predicate w.r.t. crates containing the needed resource. Since the used set of disjunctions is logically equivalent to an existential quantification, an alternative way of expressing this goal with **:existential-preconditions** (more elegant in our view) is presented as a comment in the **problem.pddl** file.

3 Problem 3.2 Approach

For this task, we proposed two different solutions: one using numeric fluents and the other modeling mathematical properties such as increments and decrements with logical predicates. The first solution

is located at *problem_2/numeric_fluents/*, while the second one at *problem_2/math_with_predicates*. The alternative to numeric fluents is proposed due to the fact that many planners do not support that requirement. An additional type **carrier** has been introduced to meet the requirement expressed in the problem definition (*"there should still be a separate type for carriers"*), while a supertype **place** has been introduced to differentiate between places where injured people are (i.e. the **loc** type) and where crates are deposited (i.e. the **warehouse** type). Introducing this last type could be very useful in case the problem would be expanded with additional deposits. Figure 2 represents the domain hierarchy of the *math_with_predicates* solution, highlighting the presence of the **quantity** type to model mathematical properties. The Fast-Downward planner has been used to generate a plan for the *math_with_predicates* version while the ENHSP-2020 planner was used for the *numeric_fluents* one.

3.1 Predicates Description

The following sections briefly show the predicates we used in the different solutions.

3.1.1 numeric_fluents

This solution has the exact same predicates as the solution for problem 1, with the **loc** type being replaced by the **place** type in localization predicates. There is no **free** predicate anymore, since the fact is modeled through the **crates_amount** function.

3.1.2 math_with_predicates

Again, the predicates remained the same as in problem 1, with the introduction of the mathematical predicates **inc** and **dec**, denoting whether an object of type **quantity** is the exact increment/decrement of another object of the same type. The **crates_amount** predicate has been introduced to keep track of how many crates are loaded onto a carrier, replacing the role of the **free** predicate of the solution to problem 1.

3.2 Actions Description

Both domain files consist of a total of four actions: **back_to_warehouse**, **move_for_delivery**, **load** and **unload**. The latter two actions resemble what shown for problem 1. The new **back_to_warehouse** action is introduced in order to cope with the requirement (*"the robotic agent [...] does not have to return to the depot until after all crates on the carrier have been delivered"*). This action handles the movement of the robotic agent from any location where injured people may reside to a warehouse and is made possible only when the carrier is effectively empty. In the **numeric_fluents** case, emptiness is checked via the **crates_amount** function while in the other via the homonym predicate. The **move_for_delivery** action handles the movement of the robotic agent from any place (be it a warehouse or a location where injured people reside) to the location of a needy person. The reason behind this splitting resides in introducing modularity in order to better manage the aforementioned constraint (robotic agent not going back to warehouses if dealing with undelivered crates). The amount of crates contained in the carrier is increased or decreased whenever the **load** or **unload** actions are called, respectively, with an upper bound of 4 crates handled with the **crates_amount** function/predicate.

3.3 Goal Description

No novelty has been introduced in the goal w.r.t. the solution of problem 1. Still, an alternative formulation in terms of *:existential-preconditions* is proposed within the comments.

4 Problem 3.3 Approach

This task is the extension of problem 3.2, with the introduction of **durative-actions**. We built our solution on top of the **math_with_predicates** solution implemented for problem 2. We chose this version because the problem stated *"the capacity of the carriers (same for all carriers) should be problem specific, [...] it should be defined in the problem file"* and the implementation with numeric_fluents was not meeting this requirement. For this assignment, we showcase that our domain file supports problems

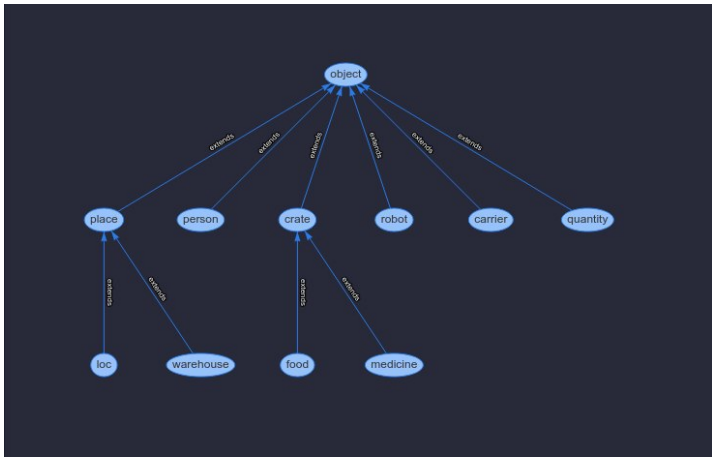

Figure 2: Domain hierarchy for problem 3.2

with both single- and multi-agent scenarios, allowing for the concurrent execution of actions. Thus, the **problem_3** folder contains two subfolders: **problem_3/single_agent** and **problem_3/multi_agent**. The domain files in both the *src* folders are identical, while the *problem.pddl* file in *multi_agent/src* defines additional objects of type **robot** and **carrier** but maintains the same initial and goal states. Since the content of **single_agent** and **multi_agent** is similar, and the latter is just a showcase for a more complicated scenario, in the rest of this section we will refer to the content of **single_agent** solely. The OPTIC Planner has been used to parse the domain-problem pair and generate a plan, so we had to refine our domain definition as well as our initial and goal states based on the limitations of the planner. As per its documentation (available here), OPTIC does not support the **:disjunctive-preconditions** and the **:negative-preconditions** requirements. Thus, we had to reformulate our goal state (which was expressed by means of disjunctions) and define complementary predicates (such as **needs** and **doesnt_need**) due to the impossibility of using negations in the goal state, while a new type **resource** was introduced to differently model the content of each crate. More details in the next sections.

4.1 Predicates Description

Most of the predicates remained unchanged. The novelties are listed below:

- the **free** predicate was reintroduced. In this case, it defines the possibility for an agent to move across places. In other terms, if the agent is free it means it is neither loading any crate onto the carrier or delivering it to someone.
- the **crate_content** predicate defines which object of type **resource** is contained within a given crate, modeling entities such as food and medicine as **resource** objects instead of subtypes of **crate**. The usefulness of this modeling will be clear when expressing the goal state.
- the **needs** and **doesnt_need** complementary predicates define whether an object of type **person** is in need of a target object of type **resource** or not. The reason behind complementary predicates resides in the usage of **doesnt_need** in the goal state and the unavailability of **:negative-preconditions** within OPTIC.

4.2 Action Descriptions

The same actions of problem 2 were leveraged. We chose a constant duration of 5 for the **back_to_warehouse** and the **move_for_delivery** actions, a duration of 2 for the **load** action (which only involves the interaction between a robot and a carrier) and a duration of 3 for the **unload** action, which involves the interaction between a robot, a carrier and a person to be served. In the **load** and **unload** actions we check that the **crates_amount** predicate for the target carrier is unchanged over all the duration of the action. This is because our bookkeeping of quantities inside the carrier is manual,

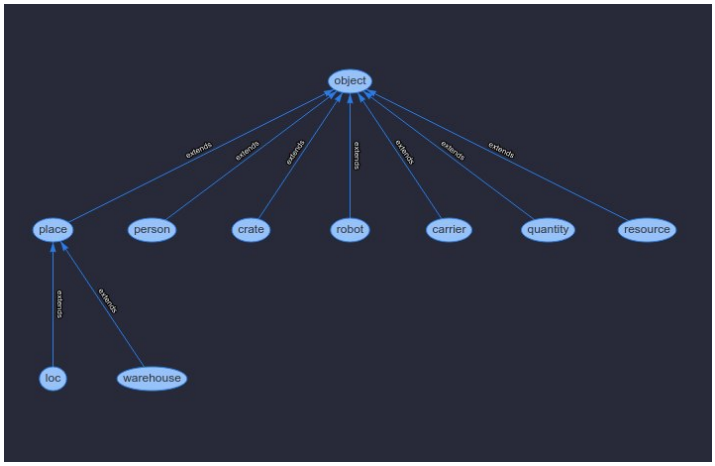

Figure 3: Domain hierarchy for problem 3.3

and we'd lose track of it if multiple robots interacted with the same carrier concurrently. Furthermore, the **unload** action now takes an additional parameter of type **resource**, allowing a robot to deliver a crate to a given person iff the given crate contains the given resource and the given resource is needed by the target person. When the crate is delivered the person is not in need of the given resource anymore. Beware that people may need more than one resource.

4.3 Goal Description

Since with OPTIC we are not able to express our goal through **:disjunctive-preconditions** nor **:negative-preconditions**, we changed the paradigm. Now, the goal is represented by a state where **every person does not need any resource**. In other terms, every person has been successfully served by a robotic agent. This translates into making sure that the **doesnt_need** predicate holds for each person w.r.t. all the available resources in the problem file. Note that if negative preconditions were available, we could express the goal in terms of the negation of the **needs** predicate in order to get rid of the **doesnt_need** predicate.

5 Problem 3.4 Approach

The final problem is implemented within the PlanSys2 infrastructure. The source code can be found in **problem_4/plansys2_assignment**. The **pddl** folder contains the reused domain file (**assignment_domain.pddl**) from the 3rd problem, while the **launch** folder contains the list of commands to be executed in the plansys2_terminal and the updated launcher file to suit our custom fake actions. The fake actions implementations can be found in the **src** folder, where a cpp file is present for each action. Since we wanted to resemble the duration specified in the domain file for each action, we implemented the system such that:

- the **back_to_warehouse** and the **move_for_delivery** last for 5s each;
- the **load** action lasts for 2s;
- the **unload** action lasts for 3s.

5.1 Custom modifications

Since the infrastructure has troubles when handling constant symbols (related Github issues here and here), a couple of tricks were introduced in the domain file in order to tackle these issues. First, we had to remove every constant symbol in the domain definition, thus defining the depot, the n0 quantity and the robotic agent via the command-line. Second, in order to meet the requirement *"the robotic agent [...] does not have to return to the depot until after all crates on the carrier have been*

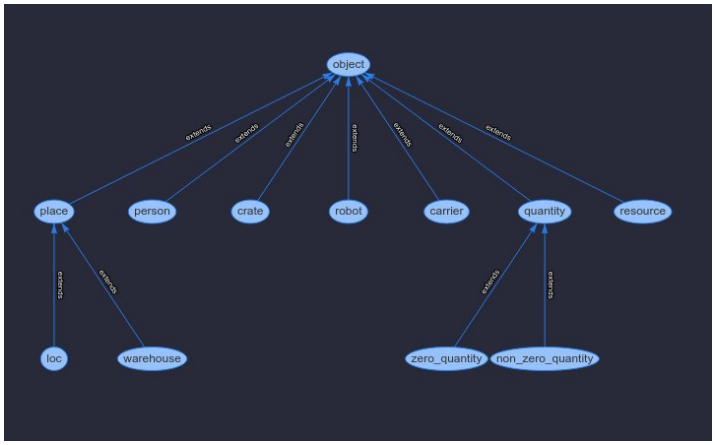

Figure 4: Domain hierarchy for problem 3.4

delivered", we modeled an additional subtyping of the **quantity** type, with the introduction of the **non_zero_quantity** and **zero_quantity** types. Consequently, the **back_to_warehouse** action can be performed only if the carrier contains a crate amount of **zero_quantity**. Figure 4 presents the overall domain hierarchy for a clearer understanding.

5.2 Installation

Additionally, we created a custom installation script, named **install.sh**, in order to ease and speed up the installation process. The script relies on the existence of a **∼/plansys2_ws** in your home directory. In case your installation resides in another folder, simply change all occurrences of **∼/plansys2_ws** with your installation folder and the script will do its job. After the installation has been performed successfully, one can run bring up the whole system via the **ros2 launch plansys2_assignment launcher.py** command.

5.3 Execution

A sample video of the whole system up and running can be found here.