Automated Planning Project

Luca Zanolo
luca.zanolo@studenti.unitn.it
University of Trento
Trento, Italy

I. Introduction

A. Overview

This report explores the design and execution of the five problems of the automated planning course project. A Python-based environment generator is employed, allowing for the creation of scenarios tailored to the specific requirements of each problem task. These scenarios are then translated into Planning Domain Definition Language (PDDL) or Hierarchical Domain Definition Language (HDDL) problems, aligning with a manually created domains for solution through various planners. This approach facilitated the development of the domain and problems for each task, also enabling a comparison of planner performances across different environment and planner configurations.

B. Report Structure

This report is organized in four principal sections. It begins with an overview of the project's file and code organization, describing the foundational structure and setup. The subsequent part offers a detailed interpretation of the scenario presented in the project assignment, clarifying the context, underlying assumptions and decisions made to accurately model the environment. This interpretation forms the basis for the domain in Problem 1 and influences the design of all subsequent problems. Following sections are dedicated to each specific task, describing the domain and problems designed for each problem of the assignment. The final portion of the report will present the results obtained with the used planners alongside a discussion on the challenges encountered throughout the project's execution.

II. Scenario

The basic scenario takes inspiration from an industrial manufacturing setting, the operating area is rapresented as a grid, having a warehouse at a specific location. This warehouse is the initial repository for all boxes and supplies necessary for operations. Additionally, a network of workstations is distributed across the grid, each requiring specific supplies to fulfill operational tasks. Robotic agents are assigned the task of transporting these supplies, packed in boxes, from the warehouse to the designated workstations.

A. Scenario Layout

The industrial environment is organized as a grid, establishing fixed locations for workstations and a central warehouse, making these entities immovable. The grid facilitates agent movement between adjacent cells, with the possibility of multiple agents sharing the same cell. The key assumptions and operational rules that guide the scenario design are summarized below.

- **Agents**
  - Agents can navigate the grid, moving between adjacent locations, adhering to the grid layout.
  - They are capable of pick up and deliver boxes at workstations, if they are at the same location of the box or the workstation.
  - Agents can fill an empty box with a supply if they are at the warehouse where the supply is stored.
  - Typically, an agent can transport one box at a time, whether empty or filled.
  - Carriers enhance an agent's capacity, allowing for the transportation of multiple boxes simultaneously.
- **Warehouse Specifications:**
  - Serves as the initial location for all supplies and boxes, with a fixed position.
  - System can generate environments with multiple warehouses, distributing boxes and supplies randomly among them.
  - Only one warehouse is allowed per location.
- **Workstation Specifications:**
  - Workstations have fixed locations, with the possibility of multiple workstations occupying the same space.
  - They may already possess certain supplies and require additional supplies of a specific type.

  - Agents deliver needed supplies in boxes to workstations, where the supplies are then unloaded to fulfill workstation needs.
  - Workstations specify the types of supplies needed, rather than specific supply items.
- **Boxes and Supplies:**
  - Boxes, initially located at the warehouse, can contain one supply item and are managed by agents for transport and delivery.
  - A box is always required to move a supply.
  - A supply can be moved only inside a box.
  - Supplies represent physical objects of predefined types, initially stored at the warehouse.
  - Supplies are classified by types, such as valves, bolts, or tools, which are used by workstations to express needs.

B. Special Considerations

Special focus is on the management and delivery of supplies to workstations. Delivering a box with the required supply to a workstation triggers a state change for the box, indicating it is unloaded at the workstation, but is no longer carried by the agent and also cannot be taken. The action of unloading the supply, making the box available for reuse, and completing the delivery requires another deliberate action by the agent.

Figure 1 shows a randomly generated area of an environment and its layout. The cell marked in red with the number six represents a location with two workstations.

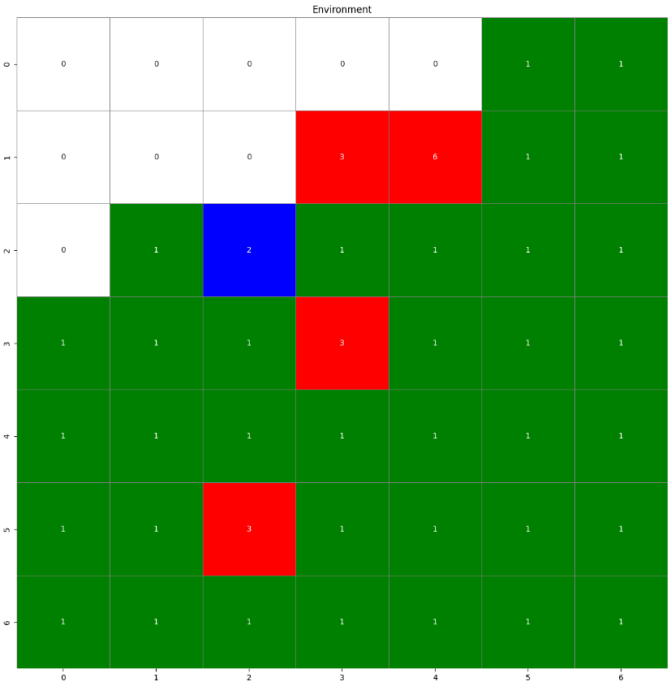

Fig. 1. Example of an environment grid where green cells are accessible and can be traversed by agents. The warehouse is located in the blue cell, and workstations are in the red cells.

III. Project Organization

A. Overview

This project is designed to automate the generation of test environments, facilitating the evaluation of different planners. The project's structure is visualized in Figure 2, providing an overview of its organization.

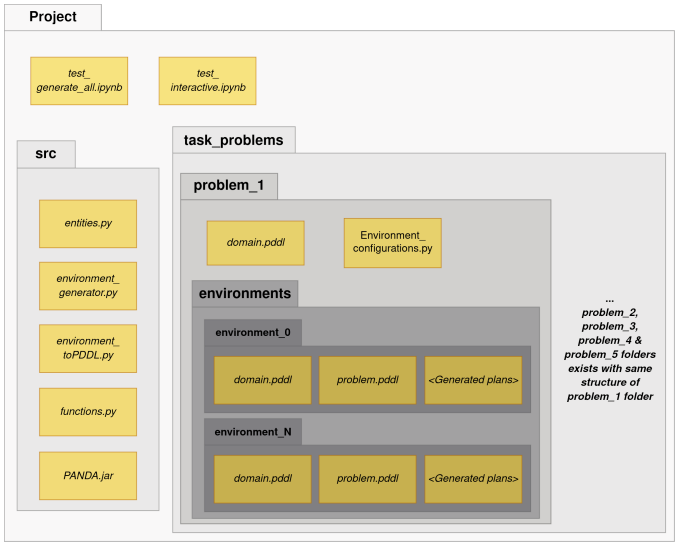

Fig. 2. Overview of the project structure and organization.

The project includes two main notebooks at its root:

- `test_interactive`: This notebook allows for the testing of a specific problem with various configurations and settings for both the environment and the planner.
- `test_generate_all`: This script generates problems and their corresponding plans based on predefined settings, populating the project folders

The `src` folder contains Python scripts crucial for generating environments, setting up problems, and retrieving plans. These scripts contains several classes that manage different aspects of environment generation.

- `environment_generator.py` Contains `Environment_generator` class which randomly generates an environment using a configuration file that specifies attributes such as the dimensions (X, Y), active cells, agents, supply types, boxes, workstations, warehouses, and carriers maximum capacity.
- `environment_toPDDL.py` Contains `Environment_toPDDL` class which generates a PDDL or HDDL problem based on the input `Environment_generator` instance.
- `entities.py` includes classes representing key objects in the environment, such as agents, boxes, supplies, workstations, warehouses, and carriers.
- `functions.py` provides utilities for retrieving plans using planutils and supports methods for output presentation.

Central to the project is the `task_problems` folder, which contains the domain and test problems for each task. The structure of Problem 5's folder is slightly different, with specifics covered in its respective section.

Each problem-specific folder within `task_problems` includes the following entities.

- An `Environments` folder containing a variety of test environments.
- A domain file that outlines the PDDL or HDDL domain for the task, and thus for all the test environments in `Environments` folder.
- An `environment_configurations` file that details the settings for predefined test environments. This configurations are always used within the `test_generate_all` notebook.

Environment folders, named from `environment_0` to `environment_N`, each hold a domain copy and a randomly generated problem instance. The `<Generated Plans>` entity of Figure 2represents where various planners' computed plans are stored.

B. Environment Generation

Below is an example outlining the generation of a test environment for Problem 1, using the available objects `Environment_generator` and `Environment_toPDDL`.

```
configuration = EnvironmentConfig(
    X=5,
    Y=5,
    active_cells=25,
    agents=1,
    supply_types=['valve', 'bolt', 'tool'],
    boxes=10,
    workstations=1,
    warehouses=1
)

environment = Environment_generator(
    configuration=configuration,
    problem_id='P1',
    verbose=True
)

environment_pddl = Environment_toPDDL(
    env=environment,
    baseline_path =
    'task_problems/problem_1',
    lang='pddl',
    verbose=True
)
```

Setting the configuration and problem ID parameters facilitates the generation of a test environment for each project problem, using a provided configuration that define the main characteristics of the generated environment. This environment is then converted into PDDL (or HDDL for certain tasks) using the `Environment_toPDDL` object. The generated problem and domain copy are saved in a newly created folder within the `Environments` directory.

After generating the environment, plan retrieval can be performed as follows:

```
request_plan(
  env_folder = environment_pddl.path OR
  <path-to-problem-domain-folder>,
  planner = <planutils-planner>,
  args = <plantuils-planner-args>,
  lang = <pddl-or-hddl>,
  args_before = <plantuils-planner-args>
)
```

This function obtains a plan with the specified planner and arguments, saving it in the newly generated environment folder. The `args_before` parameter is for planners that require a slightly different argument structure.

IV. Problem 1

This problem is based on the initial scenario explained in Section II. The domain defined here will be used as a baseline for all the domains defined for the subsequent problems. The updates and changes made will be detailed with respect to this baseline.

A. Domain

This section provides an in-depth explanation of the PDDL domain, illustrating how it addresses the scenario's requirements. Each action within the domain is designed to fulfill specific operational needs in the simulated industrial manufacturing environment.

1) Types: The domain employs specific types to model entities within the industrial scenario, specifically focusing on the following entities:

- **Agent:** Represents the robotic units responsible for transporting and managing supplies.
- **Workstation:** rapresents the workstation within the grid.
- **Warehouse:** The initial storage location for all boxes and supplies.
- **Box:** Containers that can contain supplies for transport.
- **Supply:** Physical items required by workstations to fulfill a need.
- **Location:** cell within the grid environment where various entities (agents, workstations, warehouses) can be situated.
- **Supply_Type:** Category of supplies (e.g., "valve", "bolt").

2) Predicates: Predicates define the state of the world, including the locations of various entities, the status of boxes (whether they are empty or contain supplies) and the relationships between agents, boxes, supplies, and workstations.

- **adjacent(?l1 ?l2 - location)** indicates if two locations are next to each other, enabling agent movement.

- **at(?o - locatable ?l - location)** specifies the location of locatable objects, essential for planning movements and interactions.
- **empty(?b - box)** signifies that a box is empty and ready to be filled with a supply.
- **free_arms(?a - agent)** indicates that an agent is not currently carrying a box, thus able to pick up one. This predicate highlights the agent's limitation to carrying only one box at a time.
- **box_carried_by(?b - box ?a - agent)** establishes a link between a box and the agent carrying it.
- **contains(?b - box ?s - supply)** and **is(?s - supply ?t - supply_type)** associate supplies with their types and the boxes containing them.
- **has(?ws - workstation ?t - supply_type)** and **loaded_to(?s - supply ?ws - workstation)** monitor the supply needs and their fulfillment at workstations. The first predicate is foundamental in constructing the goals for PDDL problems.

3) Actions: Actions determine how agents interact with the environment to meet objectives, addressing the scenario's requirements.

1) **move** permits an agent to move between adjacent locations, crucial for navigating the grid to deliver supplies. It requires the adjacency of the two involved locations.
2) **take_box** allows an agent to pick up a box, under the conditions that it is already carrying one and being at the same location as the box.
3) **fill_box** enables an agent to fill an empty box with a supply, assuming agent and supply are at the same location.
4) **deliver_box** facilitates the delivery of boxes to workstations by agents. This action places a box at a workstation, allowing the agent to engage in other tasks. It is often closely associated with the subsequent action for operational efficiency.
5) **deliver_supply** outlines the unloading process of a supply from a box at a workstation, ensuring the supply is appropriately allocated to the workstation. Following this action, the involved box becomes available for reuse at the workstation location. For both of these last two actions, the agent must share the same location as the workstation it interacts with.

4) Considerations: Special attention should be given to the hierarchy for `locatable` and to the distinction between `supply` and `supply_type`.

- **Locatable:** This category encompasses entities that can occupy positions in the grid, including agents, workstations, warehouses, boxes, and supplies. It enables a unified approach to determining the location of these various entities within the grid, essential for movement and action preconditions.
- **Supply vs. Supply_Type:** `supply` represents physical objects (e.g., a specific valve or bolt), essential for operational tasks at workstations. In contrast, `supply_type` abstracts these objects into categories (e.g., "valve", "bolt"), allowing workstations to generalize their needs without specifying particular items. The predicate `is(?s - supply ?t - supply_type)` effectively links physical supplies to their respective categories, simplifying the logistics of supply management by matching workstation needs with available resources. A similar mechanism will be used in Section **??** to link carriers to agents.
- **Automatic Supply Generation:** At the code level, the environment generates one supply of each type for every workstation. This strategy ensures that any combination of supply types required at each workstation can be met, offering flexibility during the random goal generation phase.

B. Problems

The predefined problems generated for this environment follow the configurations in Table I.

Setting	Environment 0	Environment 1	Environment 2
X	8	8	8
Y	8	8	8
Active cells	30	40	50
Agents	1	1	1
Boxes	10	10	10
Workstations	5	5	6
Warehouses	1	1	1
Supply types	3	4	5
Carriers	-	-	-
Goals	7	13	18

TABLE I
PREDEFINED ENVIRONMENTS CONFIGURATIONS FOR PROBLEM 1.

The `Environment_2` configuration use a single box in order to validate the boxes reusability. In general, the environments are ordered by increasing size, where the size increase in terms of number of workstation, boxes and supply types available. The goals of each environment configuration is randomly generated and consists in a conjunction of `(has ?workstation ?supply_type)` predicates.

C. Planners

For evaluating the domain and problem instances in this project task, three planners were utilized: Fast-Forward (FF), Downward, and dual-bfws-ffparser. These planners were accessed through `planutils`, a utility installed on the local system, allowing direct execution of planning commands in the form `planutils run <planner> <args>`.

- **ff:** employed as baseline for its use semplicity and effectivness.
- **Downward:** This planner allows for extensive customization through different search strategies and heuristics. It was tested with several configurations to explore its performance:
  1) (`dw_lama`) Using the LAMA-first configuration as an alias to replicate its strategy within the Downward framework.

Command arguments:
–alias lama-first

2) (dw_a*_1) Employing A* search with additive heuristics.
Command arguments:
–search astar(add(cache_estimates=true))
3) (dw_a*_2) Utilizing A* search with the context-enhanced additive heuristic (CEA), aiming to improve the accuracy of heuristic estimates.
Command arguments:
–search astar(cea(transform=no_transform(), cache_estimates=true))
4) (dw_g_1_2) An eager greedy strategy combining additive heuristics and CEA for a more diversified search approach.
Command arguments:
–search eager_greedy([add(cache_estimates=true), cea(transform=no_transform(), cache_estimates=true)])

- **Dual-bfws-ffparser:** Used to evaluate its planning capability based on width-search strategies, and because it has been proven to be very effective if compared to other planners [1]. Various configurations were experimented with to assess the impact of novelty and heuristics on planning:
  1) (dbf_1) A default setting as a reference point.
  2) (dbf_2) Enhanced delete relaxation heuristic via –use-hff, aiming to improve heuristic guidance by leveraging the Fast-Forward heuristic.
  Command arguments:
  –max_novelty 0 –use-hff 1
  3) (dbf_3) Applying the –BFWS-f5-landmarks option to integrate landmark in the search, potentially improving focus on goal-relevant actions.
  Command arguments:
  –BFWS-f5-landmarks 1
  4) (dbf_4) A polynomial search strategy using k-BFWS with maximum novelty as predefined set to 2.
  Command arguments:
  –k-BFWS 1 –max_novelty 2

The outcome for the same problem with a single planner is evaluated only in terms of the plan number of actions and the time taken to retrieve the plan. The results are showed in Table II.

Ff, used as a baseline, consistently demonstrated its effectiveness with minimal configuration, offering a reliable comparison point. With **Downward**, while configurations like dw_lama and dw_a*_1 showed promise, they also revealed that not all enhancements lead to substantial gains in performance. In fact, in larger environments, the planning time and solution quality varied significantly. **Dual-bfws-ffparser** highlighted the advantages of width-based search strategies, especially in environments with more complex constraints. Configurations emphasizing heuristic guidance (dbf_2) and novelty (dbf_4) occasionally outperformed the others.

Planner config	Environment 0	Environment 1	Environment 2
ff	64 (0.1s)	154 (0.2s)	260 (0.6s)
dw_lama	64 (0.7s)	136 (0.8s)	238 (1.0s)
dw_a*_1	71 (0.8s)	146 (1.6s)	232 (2.0s)
dw_a*_2	71 (1.0s)	142 (2.3s)	244 (1.16m)
dw_g_1_2	76 (1.1s)	162 (2.5s)	262 (4.9s)
dbf_1	64 (0.4s)	132 (0.3s)	240 (0.4s)
dbf_2	64 (0.3s)	132 (0.3s)	244 (0.4s)
dbf_3	64 (0.3s)	132 (0.3s)	236 (0.4s)
dbf_4	64 (0.3s)	132 (0.4s)	230 (0.7s)

TABLE II
PLANNERS RESULTS WITH PROBLEM 1 ENVIRONMENT PREDEFINED INSTANCES. EACH RESULT FOLLOW THIS FORMAT: [PLAN-COST (PLANNING-TIME)]

Overall, the planners' performances were relatively consistent in simpler environments, as seen in Table II. However, as the complexity of the environments increased, the differences in planning efficiency and plan quality became more evident.

V. PROBLEM 2

Problem 2 enhances the capabilities of agents by introducing carriers, each associated with a single agent and with a specific capacity. The Environment_generator class set all carriers capacities to the specified setting value in environment configuration. But the code can be easily changed (commenting and uncommenting) to interpret the carriers-capacity argument as the upper limit for randomly assigned carrier capacities, within a range starting from 2 up to the specified value. This choice were made in order to have more control on the capacity values of the carriers during tests.

A. Domain

This problem's domain builds upon the one defined in Section IV-A, extending it with new types and predicates to accommodate carriers and their capacities. The modifications regards how agents interact with carriers and manage capacities.

1) Types: The scenario introduces two new types:

- **Carrier:** Represents the carrier attached to each agent.
- **Capacity:** Denotes carrier capacity levels. For a maximum capacity of 10, as example, ten capacity objects (capacity_0 through capacity_10) are initialized to track the number of boxes a carrier can hold.

2) Predicates: To manage carrier interactions, four new predicates were added:

- **attached(?c - carrier ?a - agent):** Establishes which carrier is assigned to each agent.
- **on(?c - carrier ?b - box):** Indicates a box is loaded on a carrier.
- **carrier_capacity(?c - carrier ?c0 - capacity):** Reflects the current capacity of a carrier.
- **predecessor(?c0 - capacity ?c1 - capacity):** Assists in updating carrier capacity during take_box and deliver_box actions.

3) Actions: Actions **take_box** and **deliver_box** are modified to accommodate carrier interactions:

- Inclusion of the `attached` predicate ensures agents interact with their assigned carriers.
- The `on` predicate, added to **take_box**, signifies automatic loading of boxes onto carriers where taken from environments.
- Carrier capacity is managed via the `predecessor` predicate, adjusting capacity based on the action performed.

B. Domain - Fluents Approach

An alternative domain version, largely mirroring the one previously discussed, was developed with a difference: the incorporation of fluents to handle carrier capacities. This approach eliminates the need for the `carrier_capacity` and `predecessor` predicates. Additionally, the `total-cost` standard fluent is utilized to introduce a metric aimed at minimizing the `total-cost` value within problem configurations. The `total-cost` increases with each movement action, prompting the planner to minimize the number of movements. The goal is to potentially direct the planning process towards formulating a plan that prioritizes immediate collection of all possible boxes and supplies (given that agents typically start at the warehouse) before proceeding to delivery tasks.

To adeptly manage carrier capacities, two functions are introduced:

- **carrier-capacity ?c - carrier:** Defined within the initial environment configuration to denote the maximum capacity of a carrier.
- **carrier-load ?c - carrier:** Initially set to 0 to indicate an empty carrier. This value is adjusted upwards or downwards during the `take_box` and `deliver_box` actions, respectively. Before an agent takes a box, this function's value is checked against the carrier's maximum capacity to determine if there is space for an additional box.

The other parts of the domain logic aligns with the other domain version, maintaining consistency in the logic while integrating fluents for enhanced capacity management.

C. Problems

Three configurations were employed to assess the performance of the domain and planners for both variants of the Problem 2 domain, as detailed in Table III. The problem instances defined for each domain version represent identical scenarios and share the same objectives, facilitating a direct comparison of planner with the two domains.

D. Planners

The same planners and configurations from Section IV-C were applied with both the domain configurations, except the `Downward` planner. It is not used to solve problems using domain with fluents, since the `:fluents` requirements are not supported. Another variation regard `ff`, when tested problems with fluent instead of `ff` it's deployed `metric-ff`. The outcomes for each environment instance are detailed in Table IV.

Setting	Environment 0	Environment 1	Environment 2
X	7	7	7
Y	7	7	7
Active cells	20	30	40
Agents	2	3	4
Boxes	4	6	16
Workstations	1	2	3
Warehouses	1	1	1
Supply types	3	4	4
Carriers	2	3	4
Goals	5	8	10

TABLE III
PREDEFINED ENVIRONMENTS CONFIGURATIONS FOR PROBLEM 2.

Domain type	Planner conf.	Env. 0	Env. 1	Env. 2
no flt	ff	37 (0.1s)	69 (0.5s)	no plan (1m)
flt	metric-ff	39 (0.1s)	94 (0.4s)	no plan (1m)
no flt	dbf_1	47 (0.2s)	101 (0.8s)	122 (11.4s)
flt	dbf_1	53 (0.3s)	87 (0.5s)	110 (7.3s)
no flt	dbf_2	65 (0.3s)	188 (2.4s)	no plan (1m)
flt	dbf_2	52 (0.3s)	94 (0.3s)	104 (0.5)
no flt	dbf_3	56 (0.3s)	115 (1.2s)	108 (10.8s)
flt	dbf_3	52 (0.3s)	107 (0.5s)	117 (1.1s)
no flt	dbf_4	64 (0.7s)	115 (0.9s)	129 (2.7s)
flt	dbf_4	50 (0.4s)	100 (0.4s)	112 (1.2s)
no flt	dw_lama	50 (0.6s)	81 (0.6s)	90 (1.0s)
no flt	dw_a*_1	40 (2.4s)	no plan (1m)	no plan (1m)
no flt	dw_a*_2	40 (6.4s)	no plan (1m)	no plan (1m)
no flt	dw_g_1_2	57 (0.7s)	98 (6.8s)	100 (15s)
no flt	dbf_1	47 (0.2s)	101 (0.8s)	122 (11.4s)

TABLE IV
PLANNERS RESULTS WITH PROBLEM 2 ENVIRONMENT PREDEFINED INSTANCES. EACH RESULT FOLLOW THIS FORMAT: [PLAN-COST (PLANNING-TIME)]

Across the environments, `metric-ff` and `dual-bfws-ffparser` (dbf) configurations showed different results when applied to the fluent domain. In some instances, these configurations managed to either match or surpass their performance in the standard domain, suggesting that the introduction of fluents could enhance planning efficiency. Specifically, the dbf configurations highlighted this aspect, with settings like `dbf_2` and `dbf_4` showcasing improved outcomes in the fluent domain.

The Downward planner's varied configurations show a notable spread in performance across environments, particularly as complexity increases. Configurations like `dw_a*_1` and `dw_a*_2`, fails in larger environments to produce plans within the time constraints. The eager greedy strategy (`dw_g_1_2`) and the first configuration of dual-bfws-ffparser (`dbf_1`) generate plans across all tested environments, although with varying planning times.

These results highlight how different configurations and domains respond to the different sizes of the environments.

VI. PROBLEM 3

This problem involves adapting the domain from Problem 2 to fit a Hierarchical Task Network (HTN) approach. The main

task of delivering a supply to a workstation is now modeled using HTN's tasks and methods, maintaining the same initial conditions and goals but introducing a structured process for achieving them through hierarchical decomposition.

A. Domain

The domain retains Problem 2's types, actions, and predicates (Section V-A), enhanced with abstract tasks and methods. This tasks and methods facilitates a structured strategy to accomplish the **t-deliver_supply** goal tasks.

1) Abstract Tasks: Abstract tasks delineate the high-level behaviours that an agent could follow.

- **t-go_to**: Commands agent movement to a specific location.
- **t-take_box**: Assigns a box pickup task to an agent.
- **t-fill_box**: Directs an agent to fill a box with a specified supply.
- **t-deliver_box**: Oversees an agent's box delivery to a workstation.
- **t-deliver_supply**: The central task, mandating an agent to deliver a designated supply type to a workstation.

2) Methods: Methods decompose abastract tasks into ordered subtask sequences:

- **m-go_to Methods:** These methods detail how an agent moves to a target location, adapted from the domain examples provided in the course. Strategies include direct movement (`m-go_to`) to adjacent locations, navigating via intermediate points (`m-go_to_via`), and recognizing when the agent is already at the target (`m-go_to_imthere`) using a no-operation action (`noop`). The `noop` action is specifically designed for scenarios where no movement is required.
- **m-take_box:** Details the process for an agent to pick up a box, beginning with moving to the box's location and then executing the `take_box` action.
- **m-fill_box:** Outlines the steps for filling a box with a supply, requiring the agent to first move to the supply's location (if not already there) before performing the `fill_box` action.
- **m-deliver_box:** Describes how to deliver a box to a workstation, which includes moving to the workstation's location followed by the `deliver_box` action.
- **m-deliver_supply:** Decompose the entire supply type delivery to a workstation with the previous introduced tasks and methods. Ending with the **deliver_supply** action to finalize the delivery.

B. Problems

To evaluate the HTN domain, three environment configurations were used, detailed in Table V.

C. Planners

Panda planner, provided as a .jar object during course laboratories, was selected for Problem 3. The results for each predefined environment instance are summarized in Table VI, showing the single planner performance.

Setting	Environment 0	Environment 1	Environment 2
X	7	7	7
Y	7	7	7
Active cells	20	30	30
Agents	2	3	4
Boxes	4	6	12
Workstations	2	3	4
Warehouses	1	1	1
Supply types	3	4	4
Carriers	2	3	4
Goals	3	6	9

TABLE V
PREDEFINED ENVIRONMENTS CONFIGURATIONS FOR PROBLEM 3.

Planner config	Env. 0	Env. 1	Env. 2
panda	32 (1.1s)	53 (1.26m)	no plan (10m)

TABLE VI
PLANNERS RESULTS WITH PROBLEM 3 ENVIRONMENT PREDEFINED INSTANCES. EACH RESULT FOLLOW THIS FORMAT: [PLAN-COST (PLANNING-TIME)]

During tests, the Panda planner was deployed with various settings for the parameters `searchAlgorithm` and `heuristic`. However, the best results were achieved with a standard execution of the Panda planner without any additional arguments. By default, Panda employs `AStarActionsType(2.0)` as its search algorithm with `hhRC(hFF)` as heuristic.

VII. PROBLEM 4

Building on Problem 2, Problem 4 introduces durative actions into the domain, enabling actions with specific durations and the possibility of parallel execution. The types and predicates remain identical with those described in Section V-A for Problem 2, while action logic, preconditions and effects are adapted to account for action durations.

A. Domain

Key assumptions include:

- **move:** has a fixed duration of three. To execute, an agent must start at the `?from` location, and upon action initiation, it's considered in transit until completion. Only at the end it will be at the target location.
  The **take_box** and **fill_box** actions are assigned a duration of one time unit each, necessitating that the agent and the involved items be colocated at the action's initiation. As soon as these actions commence, updates to the box's status and the carrier's capacity are applied instantly.
  The setup of this two actions allows an agent to initiate the process of picking up a box or filling it and, concurrently, undertake a move action for example. The underlying assumption here is that the moment these actions start, the box or the supply is taken and secured by the agent's actuator, enabling the agent to move while finalizing the box's placement into the carrier.
- **deliver_box** demands the agent to remain at the workstation's location for the duration, updating the carrier's capacity at the start and the box's status at completion.

- **deliver_supply** necessitates the agent's presence at the workstation location for the entire action, making the box available for reuse immediately upon action start and satisfying the workstation's supply need only upon completion. In this way, another agent can take the initially used box, for example.

B. Problems

Three configurations, equal to those of Problem 2 (Section V-C), were employed to test the domain. The only exception regard the goals number, actually for each environment they are set to 2, 4, and 8 for `environment_0`, `environment_1`, and `environment_2`, respectively.

C. Planners

`Temporal Fast Forwar` (tfd) and `Optic` were tested. Tfd was used in its default mode, while Optic was tested with various configurations to explore different planning strategies.

- `optic_0`: Default mode with the -N argument to avoid solution optimization.
- `optic_1`: Adds -c option for earlier action tie-breaking in RPG.
- `optic_2`: Employs -e for a standard EHC instead of steepest descent.
- `optic_3`: Combines -e and -c options.

Planner performance is presented in Table VII.

Planner config	Env. 0	Env. 1	Env. 2
tfd	12 - 13 (0.6s)	34 - 47 (1.7s)	87 - 85.4 (19.9s)
optic_0	18 - 22 (0.4s)	39 - 28 (19.5s)	no plan (5m)
optic_1	15 - 13 (0.4s)	50 - 79 (4.1s)	no plan (5m)
optic_2	20 - 20 (0.3s)	39 - 28 (20.9s)	no plan (5m)
optic_3	15 - 22 (0.3s)	40 - 60 (2.4s)	no plan (5m)

TABLE VII
PLANNERS RESULTS WITH PROBLEM 4 ENVIRONMENT PREDEFINED INSTANCES. EACH RESULT FOLLOW THIS FORMAT: [PLAN-STEPS - PLAN-DURATION (PLANNING-TIME)]

VIII. PROBLEM 5

Problem 5 regard creating a ROS2 package using PlanSys2 to simulate the domain and problem scenarios described in Problem 4 (Section VII), implementing fake actions as in the PlanSys2 course tutorial.

A. Package Development

The package, located in the `p5_pkg` folder within `task_problems/problem_5`, was initiated using the `ros2 pkg create` command. This command sets up a standard package structure. Modifications were made to the `package.xml` to include necessary information and dependencies.

Each action from the domain is represented by a corresponding C++ file in `p5_pkg/src`, adhering to the template presented in the tutorials. Adjustments were primarily made to variable names and the progress value updates to simulate action execution. The `CMakeLists.txt` was then configured to ensure proper compilation of these action nodes.

B. Execution and Monitoring

Using PlanSys2's integrated Optic planner, the domain problems were solved, and the resulting plans were executed within the ROS2 framework. Execution steps, detailed in the `README` file of `p5_pkg`, allow for local replication using Docker.

To visualize the execution process, screenshots from two terminals are provided: Figure 3 captures the system state before plan execution using the `run` command in PlanSys2, while Figure 4 displays the state upon completion. These visuals help illustrate the flow of plan execution and monitoring within the PlanSys2 environment.

IX. CONCLUSION

This project has significantly enhanced my understanding of how to configure and utilize various planners to achieve complex tasks. Throughout the process, I encountered several challenges.

Initially, for Problem 2, my approach involved developing a domain based solely on fluents. However, the requirements related to fluents are not uniformly supported across all planners. This led to a pivotal adjustment in my strategy, opting for a domain that handles numerical values in a different manner. The introduction of the `predecessor` predicate to manage the sequential increase or decrease facilitate the domain compatibility with different planners.

A significant observation from the tests conducted was related to planning time. In numerous trials with larger problem instances involving more workstations and agents, sometimes a plan could not be found in an expected reasonable time. To address this issue, I tried to refine the domain to ensure it met the necessary requirements with the minimal quantity of information. Then I've experimented with adjusting planner parameters in order to achive better planning times or plan quality (evaluated only using the plan cost). These modifications sometimes led to substantial improvements in planning efficiency and, in some cases, also influenced the quality of the generated plans.

In conclusion, the effectiveness of a system in solving specific problems within a given domain relies on design of the domain and the strategic selection and configuration of planners, emphasizing that these elements are interdependent rather than sequential.

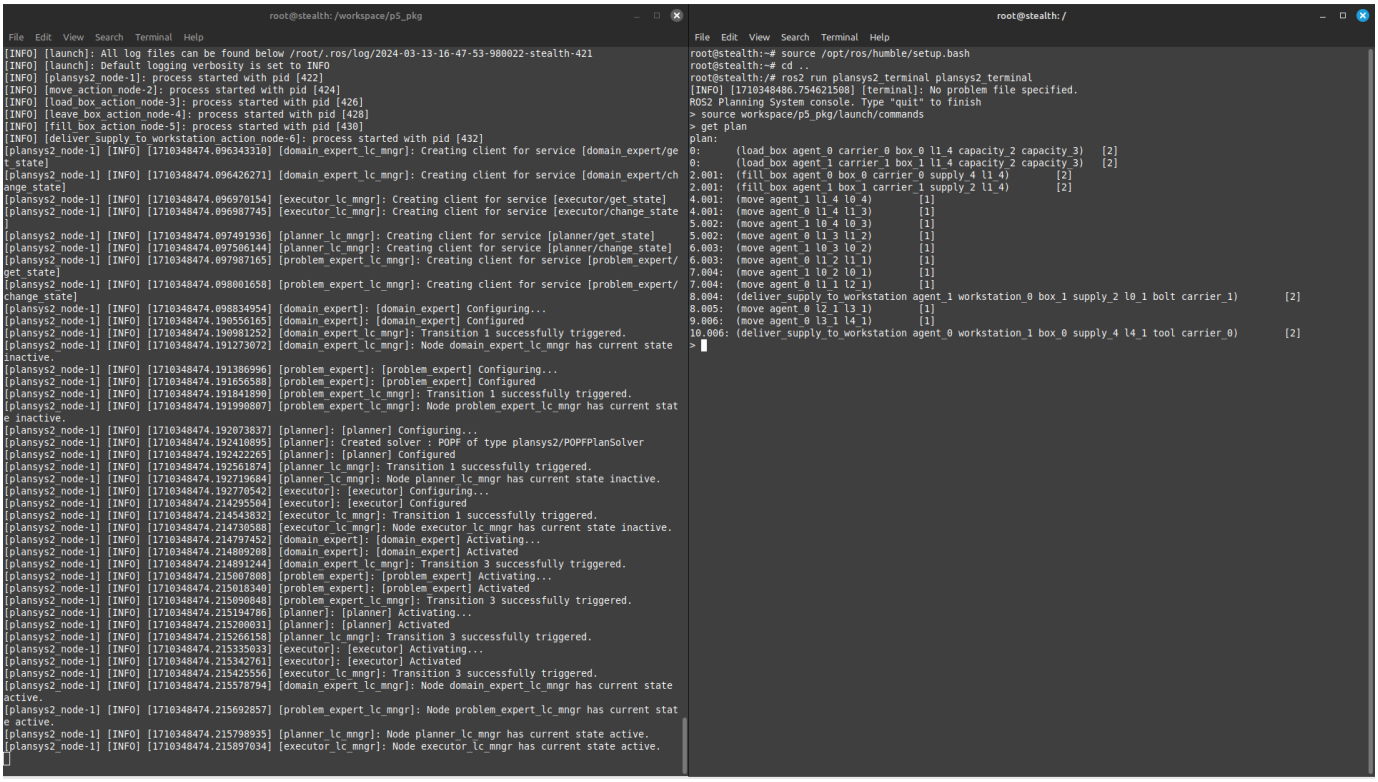

Fig. 3. Terminals at the start of the plan execution within ROS2 and Plansys2.

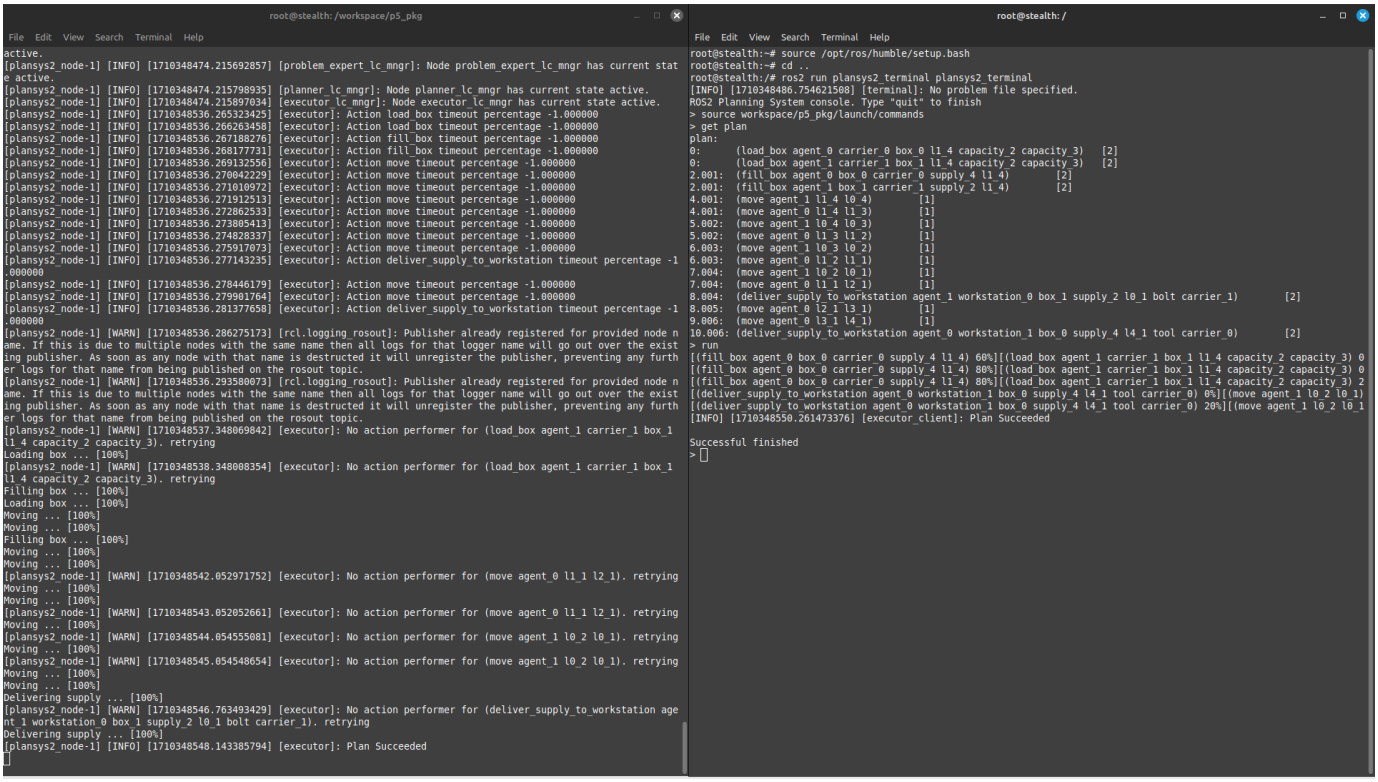

Fig. 4. Terminals at the end of the plan execution within ROS2 and Plansys2.

REFERENCES

[1] N. Lipovetzky and H. Geffner, "A polynomial planning algorithm that beats lama and ff," *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 27, no. 1, pp. 195–199, Jun. 2017. [Online]. Available: https://ojs.aaai.org/index.php/ICAPS/article/view/13822