Assignment Report

Francesco Vaccari
francesco.vaccari@studenti.unitn.it
[239927]

Report Structure

The report is divided into 5 sections, one for each Problem described in the assignment. Each section explains first the problem understanding, then illustrates the design and formal implementation, and finally shows the results of the analysis performed.

The explanation of the contents of the deliverable, the illustration of planners, and the proofs of execution can be found in the Appendix of this document.

I. Problem I

A. Problem Description

This section describes the scenario by explaining what entities are present in the environment and how a robotic agent can interact with them. The following statements are valid for Problem I, but are also to be considered valid for Problems II, III, IV and V, unless specified otherwise.

- The environment map consists of a number of locations.
- A robotic agent can only move between adjacent locations.
- To interact with elements in the environment, the agent must be in the same location of the elements. This means that no movement is required within a location.
- A robot can carry only one box.
- Each box can contain only one object.
- A robot must be carrying an empty box if it wants to pick up an object to fill the box with.
- If a robot moves between locations while carrying a box, the position of the box also changes accordingly.
- If a position of a box changes, the content of that box changes accordingly.
- Boxes and contents that are being carried cannot be interacted with by other robotic agents.
- A location can contain a number of workstations.
- Robots can drop the content of the box they are carrying to a workstation. This causes the content to be left at the workstation and also causes the box to be empty after.
- In the initial state of the problem, boxes and contents are found at a specific location called central warehouse.
- Boxes are available in limited number.
- Contents are available in unlimited quantity.

B. PDDL Domain Description

This section contains an explanation on how the problem domain has been designed and which design choices have been made.

Types:

- `location`: This type is used in the problem instance to define multiple locations.
- `workstation`: This type is used in the problem instance to define multiple workstations.
- `robot`: This type is used in the problem instance to define a robotic agent. Although the assignment description states that that only one robot is to be used, it is possible to define multiple instances.
- `box`: This type is used in problem instances to define a box.
- `content`: This type is used in problem instances to define content types. As said before, contents will be used to fill boxes. Conceptually, this definition is different from the other ones: when specifying a new content instance (ex. `bolt tool - content`), we are simply asserting that this new type of content (`bolt` or `tool`) will be available in unlimited quantity at the central warehouse, and not that a single object is created in the environment.

Constants:

- `central_warehouse`: This constant is an instance of the `location` type and is necessary in all problem definitions.

Predicates:

- `(adjacent ?l ?l)`: Used in the problem definition to specify locations the agent can move between.
- `(robot_at_location ?r ?l)`: Used to keep track of the position of a robot. In the initial problem state, robots start from the `central_warehouse`.
- `(box_at_location ?b ?l)`: Used to keep track of the position of boxes. In the initial problem state, the boxes are located at the `central_warehouse`.
- `(content_at_workstation_at_location ?c ?w ?l)`: Used to keep track of the position of a content. The fact that a workstation must be specified means that contents can only be dropped and found in workstations.
- `(workstation_at_location ?w ?l)`: Used to define which location a workstation belongs to.
- `(box_has_content ?b ?c)`: Used to keep track of what content, if any, is currently filling a box.
- `(robot_has_box ?r ?b)`: Used to keep track of which box, if any, a robot is currently carrying.
- `(robot_is_carrying ?r)`: This predicate was introduced to simplify precondition checks for certain ac-

tions that require the agent to not be carrying a box. In fact, only the robot is necessary to call this predicate.
- `(box_is_being_carried ?b)`: This predicate was also introduced to simplify precondition checks for actions that require the box to not be currently carried by other robots.
- `(box_is_full ?b)`: This predicate was also introduced to simplify precondition checks for actions that require the box to be empty.
- `(content_at_cw ?c)`: This predicate is used to add the defined content types to the `central_warehouse` so that they will be available in unlimited quantity.

Actions:

- `move`: Allows a robot to move between adjacent locations.
- `pick_up_box`: With this action a robot can pick up a box lying on the ground. Preconditions require the location of the box and of the agent to be the same.
- `put_down_box`: If a robot is carrying a box, it can execute this action. As an effect, the box will be left in the location of the robotic agent.
- `pick_up_content`: With this action the robot fills the carried empty box with some content lying on a workstation. Requirements dictate that the location of the robot and of the workstation must be the same. After executing this action, the content will no longer be at the workstation because it will be inside the now full box.
- `put_down_content`: This action has the opposite effects of the previous one, meaning that the content will no longer be inside the box, which will be empty, but will be left at the workstation. Putting down some content at a workstation that already has it, has the only effect of removing the content from the box, and not the effect of leaving two instances of the same content type at the workstation.
- `pick_up_content_at_cw`: This is a variation of the `pick_up_content` action that is necessary to allow for the unlimited availability of contents at the `central_warehouse`. To implement this functionality, preconditions require that the robot is at the `central_warehouse` location and that the predicate `(content_at_cw ?c)` is true. The main difference with the `pick_up_content` action is that the position of the content at the `central_warehouse` is not negated, thus making availability unlimited. Other preconditions and effects remain the same.
- `put_down_content_at_cw`: Just like before, this action is the opposite of `pick_up_content_at_cw`, and allows a robot to empty the carried box of its content while being at the `central_warehouse` location.

Observation 1: The problem has been modeled to allow boxes to be placed only in locations, and not in specific workstations of that location. The opposite was designed for contents: contents can be placed only in workstations, meaning that when a robot interacts with a content it will always have to specify the workstation chosen.

Observation 2: Another simplification made is that robots do not need to move within a location; this means that the robot can perform actions on any object as long as the location is the same.

Observation 3: In order to accommodate the assignment request of defining a goal as a workstation having, for example, the content `bolt`, but not the specific instance of the type bolt `bolt1` or `bolt3`, some design choices have been made. As explained before, all contents belong to the same type `content`, and only in the problem definition the specific types (`bolt`, `valve`, `tool`) are introduced. This choice has the advantage of allowing users to define by themselves new types of contents in the problem file (ex. `gear screw - content`) without needing to modify the domain file.

Observation 4: Problem goals are defined using only the predicate `(content_at_workstation_at_location ?c ?w ?l)`. The same workstation can obtain multiple contents, but they must all be of different types, for example `bolt` and `tool` but not `valve` and `valve`, which would be equivalent to obtaining a single `valve`.

Observation 5: Locations adjacency has to be specified "both ways" in the problem file, otherwise the robotic agent will be able to only move in one direction. For example, the correct definition that describes the locations `loc1` and `loc2` as adjacent is: `(adjacent loc1 loc2)` and `(adjacent loc2 loc1)`. Both predicates have to be present in the problem file.

Observation 6: In an optimal solution for problems that have goals defined as described above, certain actions will never be performed. These actions are: `pick_up_content`, `put_down_content_at_cw` and `put_down_box`. A robot would never need to pick up contents that are not requested, meaning that all contents picked up will be delivered to a workstation as defined in the problem goals. This also means that putting down contents at the `central_warehouse` is never needed. Furthermore, in an optimal solution, once the robot has picked up a box, it can achieve all goals, meaning that putting it down is never necessary.

C. Results Analysis

Three problem instances were designed: the first one is small and simple, the second one is more complex with more locations and more contents to deliver, and the third one is equal to the second one, with the only difference being the number of robots added in the formulation. The more complex problems have the objective of showing that, even if more robots would make a solution easier to find for humans, planners struggle due to the much larger search space that more robots induce.

Testing was performed using 21 combinations created by pairing a search algorithm with a heuristic. The 3 chosen search algorithms are: A*, Eager Greedy Search and Lazy Greedy Search. The 7 chosen heuristics are: Blind (as baseline), Additive, FF, Landmark-Cut, improved Pattern Database,

Causal Graph, and Context-Enhanced Additive. Additionally, LAMA 2011 has been added as a baseline configuration. All results are reported in Table I. The values reported for LAMA refer to the best solution found.

Due to the nature of the algorithm, A* is not always able to find a solution for the second problem instance. Interestingly, both Eager and Lazy Greedy search are able to find the best solution using FF and CG heuristics, taking way less time than A*. Furthermore, Greedy search is almost always able to find a solution, with the only exception being the Blind heuristic. Regarding the third problem instance, A* is unable to find even a single solution, while Greedy Search lowers the cost of some of the plans found utilizing the presence of multiple robots in the environment. Overall, looking at search times, it is possible to notice that the Blind heuristic, the LMCut heuristic and the CEA heuristic heavily impact performance in all search algorithms.

All runs have been carried out using Fast Downward [1]. Information on how to create combinations of search algorithms and heuristics can be found at [2] [3] [4] [5].

II. Problem II

The following sections illustrate only the differences with what stated in Problem I.

A. Problem Description

Two main changes were implemented in Problem II: robots need to attach a carrier before being able to carry boxes, and different kinds of robots can move in different unique ways.

The following statements explain how carriers interact with other elements:

- There can be multiple carriers in the environment and they are all initially placed in the central warehouse.
- A carrier is positioned inside a location without referring to workstations. This aspect of positioning is the same of boxes, explained in Problem I.
- Each carrier has an intrinsic property called capacity. The capacity of a carrier dictates how many boxes it can be loaded with. Content of boxes do not affect the capacity of the carrier, only the presence of the box itself does.
- Each robot can attach only one carrier. To attach a carrier, the robot needs to be in the same location of the carrier, and once attached, the carrier moves between locations with the robot, until it is detached.
- To load a box on a carrier, the robot needs to have a non-zero-capacity carrier attached. And also, just like before, the robot needs to be in the same location of the box. After the box has being loaded, the box location changes with the carrier.
- To fill a box with a content, the robot needs to have a carrier attached, and the carriers needs to have at least one empty box loaded. A robot can leave a content at a workstation if it is filling a box that is loaded on the carrier attached to the robot. Like before, the location of the robot and of the workstation must be the same.
- To sum up, robots now can only interact with boxes if they have a carrier attached, and can only interact with contents if they have a carrier attached with at least a box loaded.

In Problem II, there are 3 kinds of robots, differentiated by how they move between locations:

- *Walker*: This can be seen as the defauls kind of robot. The movement of a walker is the same of robots in Problem I, meaning it can only move between locations that are adjacent.
- *Jumper*: This kind of robot can ”jump” covering double the distance of a walker. For example, given three locations A, B and C, A adjacent to B and B adjacent to C, if the jumper is in location A it can perform one move action to go directly to location C. The necessary requirement of having an in-between location means that not all environment locations are reachable, but if a location is reachable and very far from the central warehouse, a jumper needs to perform only half the actions that a walker would need.
- *Drone*: As the name might suggest, a drone can ”fly” around the map, reaching any location in only one action. This means that locations that are not reachable normally, for example with no other location adjacent, can now be reached.

These different kinds of movement will become more interesting with the introduction of durative actions since it is possible to assign different costs, defined as time to perform an action, to the different types of movements. This allows to create problem instances with optimal solutions that require all robots to act at the same time.

B. PDDL Domain Description

The domain file designed does not use the `:numeric-fluents` requirement since it is not supported by the Fast Downward planner. A version that uses fluents was written (not tested) and is included in the deliverable, but will not be used in the results analysis.

Carrier capacities that take a numerical value have been implemented using instances of numbers (ex. `one two three - number`) that are then associated with each other through predicates to explicitly define their natural order (ex. `(number_before two one)` or `(number_after four five)`). This implementation is limited and very verbose: a capacity of, for example, 10, requires each number from 0 to 10 to be explicitly defined.

As said before, only the elements that were added, or differ from the domain specification of Problem I, are explained in this section.

Types

- `carrier`: This type allows to create carriers in the problem instance.
- `number`: This type is used to define the numerical values later used to assign a capacity to carriers.

Constants

- `zero [...] ten - number`: In addition to the `central_warehouse`, we can add in the domain file the numerical values that will be used to define capacities in the problem instance. This limits the maximum capacity to 10, but it can be later expanded by the user.

Predicates

- `walks ?r`: Since robot instances are all of the same type `robot`, we need a way to differentiate between the different kinds of robots. This predicate can define a *walker* by stating that the robot can "walk".
- `flies ?r`: This predicate can define a *drone* by stating that the robot can "fly".
- `jumps ?r`: Defines a *jumper* by stating that the robot can "jump".
- `(carrier_capacity ?c ?n)`: This predicate is used to keep track of the capacity of a carrier. In the problem instance the number used to assert this predicate will be the maximum capacity of that particular instance of carrier.
- `(increase_capacity ?n ?m)`: This predicate is used to define the ascending numerical order of the numbers created. For example, it can be used to say that the number 0 (`zero`) can be increased to 1 (`one`).
- `(decrease_capacity ?n ?m)`: The purpose of this predicate is to define the descending numerical order of the numbers created. For example, it can be used to say that the number 3 (`three`) can be decreased to 2 (`two`).
- `(carrier_at_location ?c ?l)`: This predicate is used to keep track of which location the carrier is in.
- `(robot_has_carrier ?r ?c)`: This predicate is used to keep track of which carrier a robot has currently attached, if any.
- `(carrier_has_box ?c ?b)`: This predicate is used to keep track of boxes that the carrier is currently carrying, if any. Multiple assertions of this predicate can be true at the same time if the carrier has a maximum capacity of at least 2.
- `(robot_is_attached ?r)`: Just like in Problem I, some predicates have the only purpose of simplifying the preconditions of certain actions. In this case, this predicate is used to check whether a robot has any carrier attached.
- `(carrier_is_attached ?c)`: Allows to check whether a carrier is attached to any robot.
- `(box_is_loaded ?b)`: Allows to check whether a box is currently being carried by any carrier.

Actions All actions of Problem I were modified due to the introduction of carriers. However, since their purpose has remained the same, the following list explains only the changes made.

- `move_walking`: Allows a *walker* to move between adjacent locations. The robot needs to be able to "walk" to perform this action. This property is defined using the predicate `walks ?r`.
- `move_flying`: Allows a *drone* to move between any two locations. The robot needs to be able to "fly" to perform this action. This property is defined using the predicate `flies ?r`.
- `move_jumping`: Allows a *jumper* to move between two locations if they are one location apart and they are both adjacent to the same location. The robot needs to be able to "jump" to perform this action. This property is defined using the predicate `jumps ?r`. Another precondition that helps reducing the search space requires the destination to not be the same starting location.
- `robot_attach_carrier`: To perform this action, the robot needs to be in the same location of a carrier and cannot be attached to any other carrier.
- `robot_detach_carrier`: This action allows a robot to detach the carrier it is currently attached to. If boxes are loaded on the carrier, they will still be on the carrier after the action has terminated, and the carrier will be at the same location of the robot.
- `robot_load_box`: To perform this action, the robot needs to have a carrier attached, and the carrier cannot have capacity equal to zero. After this action is performed, the box will be loaded on the carrier which will have its capacity decreased by one.
- `robot_unload_box`: This action allows robots to unload boxes from their attached carriers. After the action is performed, the box will be placed in the same location of the robot, and the carrier capacity will be increase by one.
- `pick_up_content`: This action has the same purpose of the one explained in Problem I, with the only difference being that the agent needs to have a carrier attached with at least one empty box loaded.
- `put_down_content`: This action has the same purpose of the one explained in Problem I, with the only difference being that the agent needs to have a carrier attached with at least one non-empty box loaded.
- `pick_up_content_at_cw`: This action has the same purpose of the one explained in Problem I, with the only difference being that the agent needs to have a carrier attached with at least one empty box loaded. Just like before, contents are available in unlimited quantity at the `central_warehouse`.
- `put_down_content_at_cw`: This action has the same purpose of the one explained in Problem I, with the only difference being that the agent needs to have a carrier attached with at least one non-empty box loaded.

Observation 1: When a robot wants to load a box on a carrier, we need to check if it has capacity equal to zero, and to do so we first obtain the current capacity value through `(carrier_capacity ?c ?n)`, and then we check if this value can be decreased by one unit, which can be done using `(decrease_capacity ?n ?m)`. If that smaller value exists, it will be used as the new carrier capacity after the box has been loaded. If the value does not exists then it means

that the capacity is 0 and that the action cannot be executed.

Observation 2: An implementation that makes use of `:numeric-fluents` would need to define a single **function** called `(carrier_capacity ?c)` that assigns a numerical value to a carrier to keep track of its capacity. The actions that allow the robot to load and unload boxes would use this function to check the load of the carrier and to increase or decrease its capacity.

C. Results Analysis

The same test suite of Problem I was used here. The two problem instances designed have similar complexity. The first problem instance has only a *walker* and a *jumper* defined, while the second problem instance contains only a *drone*. This choice was made to showcase the differences in movements that these different kinds of robots are able to perform.

In the first problem instance, the goal was chosen in a way that an optimal solution would require both robots to move since the workstations needing supplies are either very expensive to reach for the *walker*, or simply unreachable for the *jumper* (for the reason explained in the problem description).

In the second problem instance, the environment comprises of locations whose adjacency has not been defined. This does not affect the *drone* since its ability to fly does not require locations to be adjacent to move between them. Since the size of the search space is reduced with only one robot, more goals were defined to counter this effect.

Results are shown in Table II. As we can see, A* is not always able to find the optimal solution in problem instance 1. This phenomenon, that did not occur in the Problem I's analysis of results, happens with the heuristics that are not admissible, with the exception of the FF heuristic. However, only inadmissible heuristics are able to find a solution in problem instance 2, while all other configurations of A* fail to terminate. Regarding the configurations with the Greedy search algorithm, the iPDB heuristic and the Blind heuristic are unable to find a solution in problem instances 1 and 2 respectively. Interestingly, in problem instance 1, the configuration with the Blind heuristic, which should act as the baseline for all other heuristics, is able to find the optimal solution, which was otherwise not found by the other configurations. Similarly to what found in the analysis of Problem I, the heuristics Blind, LMCut and CG are the ones that most heavily impact performance.

III. Problem III

In problem III no new elements are introduced and no changes are made to existing objects and to how the robots interact with them.

A. HDDL Domain Description

The major difference in Problem III is the usage of a Hierarchical Task Network (HTN) approach [6]. This approach defines the goal as a series of tasks to achieve, with the possibility of specifying an order in which the tasks have to be completed. Each task can be reached by executing methods defined in the domain file. The tasks implemented represent something that a robot might want to achieve, for example getting a carrier or making a delivery.

Tasks:

- `(move (?r ?l))`: This task is used by all types of robots to move in the environment.
- `(get_carrier (?r ?car))`: This task is used to let robots attach carriers
- `(get_box (?r ?b))`: This task is used when a robot wants to load a box. Only the robot and the wanted box are specified in the parameters, meaning that the carrier necessary to complete the task is chosen by the planner.
- `(get_content (?r ?con))`: This task is used when a robot wants to obtain some content. Only the robot and the content are specified in the parameters, meaning that both the carrier and the box are chosen by the planner.
- `(deliver (?con ?w))`: This is the task used in the problem goal definition. Robot, carrier, and box are all chosen by the planner, making the search very computationally expensive but also making the goal independent from the specific instances defined.

Each task is completed by executing the methods that implement it. Methods designed can recursively call themselves, or call other methods, depending on which state the environment is in. The state of the environment that conditions which method is chosen is explained in the following list, as well as the subtasks that implement the method itself.

Methods for `(move (?r ?l))`:

- `move_0`: The robot is already at the location it wants to go to, meaning that no action is performed.
- `move_walking_1`: The robot is a *Walker* and the location it wants to reach is adjacent to the location it is currently in. In this situation the robot performs one `move_walking` action.
- `move_walking_2`: The robot is a *Walker* and the location it wants to reach is not adjacent to the location it is currently in. In this situation the robot moves to an adjacent location and then calls the same task it is trying to implement. Doing so, the planner is able to find the path that allows the robot to get to its destination.
- `move_flying_1`: The robot is a *Drone*, meaning it can move between any two locations by performing the `move_flying` action.
- `move_jumping_1`: The robot is a *Jumper* and the location it wants to reach is one "jump" away. In this situation the robot performs one `move_jumping` action.
- `move_jumping_2`: The robot is a *Jumper* but the location is not one "jump" away. In this situation the robot performs one `move_jumping` action and then calls the `move` task.

Methods for `(get_carrier (?r ?car))`

- get_carrier_0: The robot has the carrier already attached which means that no subtasks are performed.
- get_carrier_1: The robot has a carrier attached but it is not the one wanted. The robot first needs to detach the unwanted carrier and only then can attached the correct one.
- get_carrier_2: The carrier is not attached to any robot. In this situation the robot moves to the location of the carrier, which might be already the same of the robot, and attaches it with the robot_attach_carrier action.
- get_carrier_3: The carrier is attached to another robot. In this situation the other robot is instructed to detach it, and then the get_carrier task is called.

Methods for (get_box (?r ?b))

- get_box_0: The box is already loaded to the attached carrier. In this case no subtasks are performed.
- get_box_1: The box is on the ground and the robot has a non-zero-capacity carrier attached. In this situation the robot moves to the location of the box and loads it with the robot_load_box action.
- get_box_2: The box is on the ground but the robot has no carrier attached. In this case the robot needs to get a carrier with free space available, which is done with the get_carrier task.
- get_box_3: The box is on the ground but the robot has a zero-capacity carrier attached. In this situation, the robot gets another carrier with free space (get_carrier task) and then calls the get_box task.
- get_box_4: The box is on the ground but the robot has a zero-capacity carrier attached. An alternative to detaching the carrier and looking for a new one is unloading a box to make space for the wanted one. This is achieved by performing a robot_unload_box action and then calling the get_box task to load the correct one.
- get_box_5: The box is loaded on a carrier not attached to the robot. In this case the task get_carrier is called.
- get_box_6: The box is loaded on a carrier not attached to the robot. In this situation the other robot is instructed with dropping the box from its carrier, and then the get_box task is called to retrieve it from the ground.

Methods for (get_content (?r ?con))

- get_content_0: The content is already in a box loaded on the carrier attached. No subtasks are executed.
- get_content_1: The robot has an empty box loaded on the carrier attached. In this case the robot moves to the central warehouse and picks up the wanted content with the pick_up_content_at_cw action.
- get_content_2: The robot has a carrier attached but with no empty boxes. In this situation the robots first obtains an empty box through the get_box task and then calls the get_content task.
- get_content_3: The robot has no carrier attached. In this case, first the get_carrier task is called and then the get_content task is called.
- get_content_4: The wanted content is in a box not currently being carried by the robot. In this situation, the get_box task is called, possibly even stealing the box or the carrier with the box by other robots.

Methods for (deliver (?con ?w))

- deliver_0: Now that all other methods and tasks are implemented, defining how to perform a delivery is quite straightforward. First, the task get_content is called to obtain the content needed for the delivery, then the task move is called to make the robot reach the destination, and then the action put_down_content is executed to complete the goal.
- deliver_1: To allow the planner to find optimal solutions, before starting a delivery, it is possible to pick up some other content through this method. The subtasks called are, in order, get_content and deliver. It is the planner that decides whether to pick some content, other than the requested one, to use it for future deliveries once the current one is achieved.

Observation 1: Constraints defined in all methods are crucial to reduce the size of the search space. This is especially useful since it is the planner that decides which robots, carriers and boxes are used when making a delivery.

Observation 2: Another way to reduce the search space size is assuming that contents, when not inside of boxes, can only be found at the central warehouse. This is a reasonable assumption if we consider that, in optimal solutions, if a content is at a workstation, then it was brought there to satisfy a goal, and thus should not be picked up.

B. Results Analysis

The configuration of Panda [7] used is the default one, which uses A* with the FF heuristic in a relaxed planning approach. Attempts to use other configurations available, or to use other search algorithms in combination with different heuristics, have failed.

Another problem encountered relates to how Panda defines the cost of a solution. Conceptually, a plan cost should be only defined by how many actions it consists of. However, Panda considers methods as a part of solutions and assigns them the same cost (1) that is assigned to actions. For this reason, a solution found does not necessarily try to minimize the number of actions that achieve the tasks given because Panda adds in the number of methods that implement those tasks.

Furthermore, due to the implementation having a lot of methods with the possibility of recursive calls, the search space is very large. For example, it was not possible to test on the same problem instances used in the analysis of Problem II (adapted to HTN) because no solution was found after several minutes of search.

For all these reasons, the only comparison possible consists in creating multiple problem instances where the type and

number of robots included are the only differences. For all goals a strict order is specified to simplify the search for a solution.

- *Problem Instance 1*: The environment consists of 6 locations forming a line so that location 6 is the farthest from the central warehouse. There are only two carriers, both with capacity one, and there are also only two boxes. The goal consists of two deliveries, one at location 4 and one at location 6.
- *Problem Instance 2*: This problem has the same definition of the first instance, with one exception regarding movement. The adjacency of locations is "one-way" only, meaning that, for example, if a robot reaches location 4, it can only move to locations 5 and 6, and con no longer go back to locations already visited. Number of carriers, their capacity, the goal, and the number of boxes are the same.
- *Problem Instance 3*: This instance is less complex than the first, with only 4 locations positioned in a straight line. The goal is similar, defining again only two delivers in locations 2 and 4. Number of carriers, their capacity, and the number of boxes are the same of Problem Instance 1.
- *Problem Instance 4*: Just like in the second instance, the adjacency constraint is added to Problem Instance 3 to disallow robots to come back to the central warehouse. Number of carriers, their capacity, the goal, and the number of boxes are the same.

Results are reported in Table III. The plan cost indicates the number of actions in the plan, and the number in parenthesis is the total cost of the plan including the methods, as it was reported by Panda.

First of all, since two deliveries are required, all problems containing one single robot and the "one-way" constraint are not satisfiable because all carriers available have capacity equal to one. This is true for *Jumpers* and *Walkers* but not for *Drones* since they do not need locations to be adjacent to move between them. The "one-way" constraint was introduced in the problems designed to force both robots to perform actions. Otherwise, only one of the two robots available would make deliveries. This is proved by the fact that all solution costs remain the same, or worsen, with the constraint put in place.

Consider the following situation: `Drone_1` has just delivered a supply and is still in the delivery location, to perform another delivery it needs to go back to the central warehouse, fill up the now empty box which is still loaded on the carrier, and then move to the new location to perform the second delivery. To sum up, `Drone_1` has to perform 4 actions before completing the second delivery. Now let's consider `Drone_2` from the same initial state explained above. `Drone_2` is located at the central warehouse and needs to attach a carrier, load an empty box and only then can load the content, move to the delivery location and put down the content. This results in 5 actions for `Drone_2` to perform the second delivery. This proves that if any of the two delivery locations is less than three moving actions away from the central warehouse, then the goal can be achieved by only one robot. This means that making a problem that needs 2 drones to be solved is impossible, and that problems that require 2 jumpers to move are very computationally expensive to solve.

Going back to the results in Table III, it is possible to notice that the presence of a *Walker* slows down the search process probably due to the computational burden that the walking movement introduces. To a lesser extent, the same is true for *Jumpers*.

IV. Problem IV

Problem IV introduces the temporal aspect to the same environment and interactions of Problem II, changing the definition of solution cost and allowing different robots to perform actions at the same time.

A. Temporal PDDL Domain Description

The definition of temporal planning domains [8] introduces durative actions that replace the actions seen in last Problems. A durative action is defined with preconditions, effects, parameters, and a duration. For each precondition it is possible to define the interval of the action in which the statement must be true for the action to be executable. The same thing can be done also for effects, with the interval specifying when the statement will be made true. In this context, intervals can be used to render actions of the same robot nonsimultaneous, as it would happen in reality. Each durative action is also defined by a duration, which is the amount of time that the robot will spend executing it.

Table IV shows exactly which actions a robot can perform at the same time. Actions have been grouped into categories for a simpler explanation. Movement is mutex with all other actions because interacting with objects requires the location to remain the same for the entire duration of the interaction. Attaching and detaching a carrier cannot be executed while interacting with boxes and contents because the carrier attachment is a precondition that lasts for the entirety of the action. Making robots unable to interact with boxes (load or unload) at the same time has been a choice with the objective of showing how a mutex can be implemented. In fact, the same could have been done for interactions with contents. Although interactions with boxes and contents can be performed simultaneously, the interactions cannot affect the same object instances together. We can imagine the robot having a single robotic arm that loads and unloads boxes, and having multiple claw crane arms that fill and empty loaded boxes.

Two planners were used to solve temporal problems: Optic and TFD. Both planners support `:numeric-fluents` which allows for a better implementation of carrier capacities. However, Optic does not support `:negative-preconditions`, meaning that some predicates had to be removed and other had to be introduced. Furthermore, the predicate `(mutex_load_box ?r)` was added to not allow robots to perform multiple `robot_load_box` or `robot_unload_box` actions in parallel.

	Problem Instance 1		**Problem Instance 2**		**Problem Instance 3**	
	Plan Cost	Search Time	Plan Cost	Search Time	Plan Cost	Search Time
LAMA 2011	21	0.000575s	45	0.171199s	44	1.706762s
A* + Blind	21	0.730149s	/	/	/	/
A* + ADD	21	0.001519s	45	0.399645s	/	/
A* + FF	21	0.056703s	/	/	/	/
A* + LMCut	21	0.697055s	/	/	/	/
A* + iPDB	21	0.078287s	/	/	/	/
A* + CG	21	0.002355s	45	0.024758s	/	/
A* + CEA	21	0.002446s	45	0.761417s	/	/
E-Greedy + Blind	21	0.591114s	/	/	/	/
E-Greedy + ADD	21	0.000583s	63	6.522030s	44	0.038679s
E-Greedy + FF	24	0.000812s	45	0.004566s	43	0.065613s
E-Greedy + LMCut	21	0.005512s	49	0.154322s	62	9.790830s
E-Greedy + iPDB	21	0.000302s	46	0.001590s	43	0.049070s
E-Greedy + CG	22	0.000563s	45	0.005256s	44	0.032957s
E-Greedy + CEA	21	0.000940s	63	11.781500s	44	0.087985s
L-Greedy + Blind	21	0.793403s	/	/	/	/
L-Greedy + ADD	21	0.000379s	65	5.344640s	56	0.252156s
L-Greedy + FF	21	0.000466s	45	0.002270s	45	0.006875s
L-Greedy + LMCut	21	0.003134s	56	0.073553s	64	31.3606s
L-Greedy + iPDB	21	0.000440s	46	0.002488s	43	0.147031s
L-Greedy + CG	21	0.000404s	45	0.026656s	59	0.033109s
L-Greedy + CEA	21	0.000498s	65	8.979340s	54	0.476814s

TABLE I

SOLUTION COSTS AND SEARCH TIMES FOR THE CONFIGURATIONS TESTED ON THE 3 PROBLEM INSTANCES DESIGNED.

	Problem Instance 1		**Problem Instance 2**	
	Plan Cost	Search Time	Plan Cost	Search Time
LAMA 2011	16	5.76371s	27	1.98200s
A* + Blind	16	12.90460s	/	/
A* + ADD	21	1.03346s	28	47.034s
A* + FF	16	0.32319s	/	/
A* + LMCut	16	13.723s	/	/
A* + iPDB	16	1.52817s	/	/
A* + CG	21	1.16998s	27	19.4313s
A* + CEA	21	3.00248s	/	/
E-Greddy + Blind	16	10.5488s	/	/
E-Greddy + ADD	23	0.19945s	32	0.12348s
E-Greddy + FF	24	0.22855s	32	0.02581s
E-Greddy + LMCut	45	1.28056s	29	1.07232s
E-Greddy + iPDB	/	/	33	0.00298s
E-Greddy + CG	22	0.26637s	32	0.15941s
E-Greddy + CEA	23	0.35178s	32	0.36180s
L-Greddy + Blind	16	18.1872s	/	/
L-Greddy + ADD	32	0.03101s	33	0.15398s
L-Greddy + FF	32	0.00984s	33	0.00756s
L-Greddy + LMCut	47	1.07044s	31	0.79080s
L-Greddy + iPDB	/	/	38	0.00687s
L-Greddy + CG	22	0.22352s	42	0.18905s
L-Greddy + CEA	87	0.12037s	40	0.00831s

TABLE II

SOLUTION COSTS AND SEARCH TIMES FOR THE CONFIGURATIONS TESTED ON THE 2 PROBLEM INSTANCES DESIGNED.

	Problem Instance 1		**Problem Instance 2**		**Problem Instance 3**		**Problem Instance 4**	
	Plan Cost	Search Time	Plan Cost	Search Time	Plan Cost	Search Time	Plan Cost	Search Time
1 Walker	20 (45)	2027ms	/	/	14 (33)	134ms	/	/
1 Jumper	13 (31)	64ms	/	/	10 (25)	22ms	/	/
1 Drone	9 (23)	45ms	9 (23)	62ms	9 (23)	23ms	9 (23)	31ms
2 Walkers	20 (45)	59591ms	/	/	14 (33)	558ms	14 (38)	17675ms
2 Jumpers	13 (31)	247ms	13 (36)	17656ms	10 (25)	152ms	11 (32)	2986ms
2 Drones	9 (23)	368ms	9 (23)	389ms	9 (23)	186ms	9 (23)	239ms
1 Walker + 1 Jumper	13 (31)	402ms	15 (40)	71687ms	10 (25)	141ms	12 (34)	8938ms
1 Walker + 1 Drone	9 (23)	267ms	9 (23)	300ms	9 (23)	211ms	9 (23)	219ms
1 Jumper + 1 Drone	9 (23)	322ms	9 (23)	316ms	9 (23)	147ms	9 (23)	145ms

TABLE III

SOLUTION COSTS AND SEARCH TIMES FOR THE 4 PROBLEM INSTANCES DESIGNED WITH THE 4 STARTING CONFIGURATIONS OF ROBOTS.

The following predicates, which simplify precondition checks for some actions, were introduced to comply with the no-negative-preconditions requirement:

- `(robot_is_not_attached ?r)`
- `(carrier_is_not_attached ?c)`
- `(box_is_not_loaded ?b)`
- `(box_is_not_full ?b)`

The function `(carrier_capacity ?c)`, which is used like in the following example `(= (carrier_capacity carrier) 2)`, was added since numeric fluents are supported in the temporal planners employed for the results analysis.

Table V shows the duration values chosen for actions in Problem IV. The duration of movement actions was chosen to be so long to force the different robots defined to make deliveries simultaneously. Moreover, these duration values make solutions interesting: the Jumper can cover double the distance of a *Walker* while spending less time doing so, but is unable to reach all locations; the *Drone*, instead, can move to any location in one action but the execution takes a lot of time, making it more efficient than a *Walker* only if the location is at least 4 steps away.

B. Results Analysis

Two planners were used: Optic [9], which uses Weighted A* with relaxed plan length as heuristic [10], and TFD [11], which uses A* with an adaptation of the context-enhanced additive heuristic. Both were run on the three problem instances designed using the default configurations.

The three problem instances contain many locations and objects and differ only in the goals defined: Instance 1 needs three deliveries completed, Instance 2 needs four, and Instance 3 needs six. Results are shown in Table VI. Reported solutions for Optic are the best found in the 20 minutes time limit given. The plan costs reported indicate the total time spent to achieve the goals, while the number in parenthesis is the number of actions contained in the plan. TFD is always able to find a much better solution than Optic, while also spending a lot less time searching for it, which is quite impressive considering that Optic is able to improve its solution by continuing the search after finding the first one.

V. Problem V

ROS2 Planning System (PlanSys2) [14] is a PDDL-based planning system. It is implemented in ROS2, and allows to plan and run solutions for temporal planning problems. The domain is the same of Problem IV, with durative actions implemented as fake actions as explained in [15]. The duration of fake actions has been scaled down to a tenth of the original value, which, combined with a 0.1 progress update, makes plans execute at the original intended speed.

Observation 1: The requirement `:numeric-fluents` should be supported since PlanSys2 uses Optic at its core, but when using this requirement the system was not working. Once the fluents were removed and the old implementation of carrier capacities used in Problem II was reintroduced, the system worked flawlessly. For this reason, the domain and problems used in the results do not employ `:numeric-fluents`.

Observation 2: Another problem encountered was that constants defined in the domain file were not recognized. To solve this issue, all problem instances also contain the instantiation of the `central_warehouse` and of all the capacity numerical values.

A. Results Analysis

Since PlanSys2 uses the Optic planner to find solutions before running them, it is possible to compare the different plans obtained in the same three problem instances used in the results analysis of Problem IV. Results are shown in Table VII. Unfortunately, PlanSys2 does not output a search time, meaning that it is not possible to compare these values. Moreover, for problem instance 3, the planner wasn't able to find a solution probably due to a time limit implemented in the system. However, the solutions found have lower or equal cost in both cases. As proof, Figures 1, 2 and 3 show the solutions found.

References

[1] "HomePage - Fast Downward Homepage." Accessed August 5, 2024. https://www.fast-downward.org.

[2] "IpcPlanners - Fast Downward Homepage." Accessed August 5, 2024. https://www.fast-downward.org/IpcPlanners.

[3] "Doc/SearchAlgorithm - Fast Downward Homepage." Accessed August 5, 2024. https://www.fast-downward.org/Doc/SearchAlgorithm.

[4] "Doc/Evaluator - Fast Downward Homepage." Accessed August 5, 2024. https://www.fast-downward.org/Doc/Evaluator.

[5] "PlannerUsage - Fast Downward Homepage." Accessed August 5, 2024. https://www.fast-downward.org/PlannerUsage.

[6] Höller, Daniel, Gregor Behnke, Pascal Bercher, Susanne Biundo, Humbert Fiorino, Damien Pellier, and Ron Alford. "HDDL–A Language to Describe Hierarchical Planning Problems." arXiv preprint arXiv:1911.05499 (2019).

[7] https://www.uni-ulm.de/en/in/institute-of-artificial-intelligence/research/software/panda/

[8] Fox, Maria, and Derek Long. "PDDL2. 1: An extension to PDDL for expressing temporal planning domains." Journal of artificial intelligence research 20 (2003): 61-124.

[9] "Planning at KCL." Accessed August 5, 2024. https://nms.kcl.ac.uk/planning/software/optic.html.

[10] Benton, J., Amanda Coles, and Andrew Coles. "Temporal planning with preferences and time-dependent continuous costs." In Proceedings of the International Conference on Automated Planning and Scheduling, vol. 22, pp. 2-10. 2012.

[11] "TFD · Temporal Fast Downward," March 1, 2012. Accessed August 5, 2024. https://tfd.informatik.uni-freiburg.de/index.html.

[12] Eyerich, Patrick, Robert Mattmüller, and Gabriele Röger. "Using the context-enhanced additive heuristic for temporal and numeric planning." In Towards Service Robots for Everyday Environments: Recent Advances in Designing Service Robots for Complex Tasks in Everyday Environments, pp. 49-64. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[13] Muise, Christian. "Planutils." PyPI, August 3, 2024. Accessed August 5, 2024. https://pypi.org/project/planutils/.

[14] Rey Juan Carlos University, and Francisco Martín. "ROS2 Planning System — ROS2 Planning System 2 1.0.0 Documentation." Accessed August 5, 2024. https://plansys2.github.io/index.html.

[15] Rey Juan Carlos University, and Francisco Martín. "Tutorials — ROS2 Planning System 2 1.0.0 Documentation." Accessed August 5, 2024. https://plansys2.github.io/tutorials/index.html.

	Movement	Interaction with carriers	Interaction with boxes	Interaction with contents
Movement	**No**	**No**	**No**	**No**
Interaction with carriers	**No**	**No**	**No**	**No**
Interaction with boxes	**No**	**No**	**No**	**Yes**
Interaction with contents	**No**	**No**	**Yes**	**Yes**

TABLE IV
PARALLEL EXECUTION COMPATIBILITY OF DURATIVE ACTIONS DEFINED.

Action Name	**Duration**
`move_walking`	20
`move_jumping`	30
`move_flying`	70
`robot_attach carrier`	3
`robot_detach carrier`	3
`robot_load box`	2
`robot_unload box`	2
`pick_up_content`	1
`put_down_content`	1
`pick_up_content_at_cw`	1
`put_down_content_at_cw`	1

TABLE V
DURATION VALUES CHOSEN FOR THE DURATIVE ACTIONS IMPLEMENTED IN PROBLEM IV

	Problem Instance 1		**Problem Instance 2**		**Problem Instance 3**	
	Plan Cost	Search Time	Plan Cost	Search Time	Plan Cost	Search Time
TFD	128.132 (17)	0.036s	89.094 (21)	0.035s	225.244 (46)	53.142s
Optic	214.004 (25)	0.81s	227.011 (24)	207.54s	501.014 (37)	1101.18s

TABLE VI
PLAN COSTS (NUMBER OF ACTIONS) AND SEARCH TIMES OBTAINED BY RUNNING DEFAULT CONFIGURATIONS OF OPTIC AND TFD ON THE THREE PROBLEM INSTANCES DESIGNED.

	Problem Instance 1		**Problem Instance 2**		**Problem Instance 3**	
	Plan Cost	Search Time	Plan Cost	Search Time	Plan Cost	Search Time
TFD	128.132 (17)	0.036s	89.094 (21)	0.035s	225.244 (46)	53.142s
Optic	214.004 (25)	0.81s	227.011 (24)	207.54s	501.014 (37)	1101.18s
PlanSys2	94.004 (19)	/	89.009 (21)	/	/	/

TABLE VII
PLAN COSTS (NUMBER OF ACTIONS) AND SEARCH TIMES OBTAINED BY RUNNING DEFAULT CONFIGURATIONS OF OPTIC, TFD AND PLANSYS2 ON THE THREE PROBLEM INSTANCES DESIGNED.

APPENDIX A
DELIVERABLE STRUCTURE AND CONTENTS

The deliverable consists of five folders, one for each Problem. The first four folders contain:

- the `domain` file.
- the `problem` files: these are the problem instances used in the analysis sections.
- the `commands` file, which contains the terminal commands run to obtain the results explained in the analysis sections.
- the `logs` directory, which contains the terminal outputs that the commands produced while executing.

In the `Problem2` folder there is also the domain file defined with `:numeric-fluents`, although it does not work with the planner used in that section.

The `Problem5` folder contains the docker configuration file used to setup PlanSys2, and the folder `package` which contains what is needed to run Problem V. The instructions on how to build and run the package provided are in the `README` file inside the `package` folder. Moreover, since the package is a template from [15], comments have been added to all files to indicate which specific lines or sections of code have been added or modified.

APPENDIX B
ABOUT PLANNERS

The assignment was completed on a Macbook with the M1 chip, which made things much more difficult. Luckily, both Fast Downward and Panda, obtained from [1] and [7], were able to run natively. However, to run Optic, TFD and PlanSys2, the use of Docker was necessary. Optic and TFD were run through planutils, which can be used in a docker container following the instructions at [13]. PlanSys2 was instead run on a custom Docker configuration (provided by Professor Roveri) which can be found in the `Problem5` folder in the deliverable.

APPENDIX C
PROOF OF EXECUTION

As a proof of execution, the command outputs that produced the results reported in the analysis sections have been saved in the `logs` folders for all problems except Problem V. For this Problem, screenshots were taken as proof, and are here reported in Figures 1, 2, 3 and 4.

```
root@docker-desktop:/home/package# ros2 run plansys2_terminal plansys2_terminal
[INFO] [1722783225.211112215] [terminal]: No problem file specified.
ROS2 Planning System console. Type "quit" to finish
> source launch/problem_1
> get plan
plan:
0:      (move_jumping jumper central_warehouse loc1 loc2)       [30]
0:      (robot_attach_carrier drone carrier1 central_warehouse) [3]
0:      (robot_attach_carrier walker carrier2 central_warehouse)        [3]
3.001:  (robot_load_box drone carrier1 box1 central_warehouse two one)  [2]
3.001:  (robot_load_box walker carrier2 box2 central_warehouse two one) [2]
5.002:  (pick_up_content_at_cw drone carrier1 box1 valve)       [1]
5.002:  (pick_up_content_at_cw walker carrier2 box2 tool)       [1]
6.003:  (move_flying drone central_warehouse loc5)      [70]
6.003:  (move_walking walker central_warehouse loc1)    [20]
26.004: (put_down_content walker carrier2 box2 loc1 tool wor1)  [1]
27.005: (move_walking walker loc1 central_warehouse)    [20]
30.001: (move_jumping jumper loc2 loc1 central_warehouse)       [30]
47.006: (pick_up_content_at_cw walker carrier2 box2 valve)      [1]
48.007: (robot_detach_carrier walker carrier2 central_warehouse)        [3]
51.008: (move_walking walker central_warehouse loc1)    [20]
60.002: (robot_attach_carrier jumper carrier2 central_warehouse)        [3]
63.003: (move_jumping jumper central_warehouse loc1 loc2)       [30]
76.004: (put_down_content drone carrier1 box1 loc5 valve wor5)  [1]
93.004: (put_down_content jumper carrier2 box2 loc2 valve wor2) [1]
```

Fig. 1. Solution found by PlanSys2 for problem instance 1.

```
root@docker-desktop:/home/package# ros2 run plansys2_terminal plansys2_terminal
[INFO] [1722783261.859548218] [terminal]: No problem file specified.
ROS2 Planning System console. Type "quit" to finish
> source launch/problem_2
> get plan
plan:
0:      (robot_attach_carrier drone carrier1 central_warehouse) [3]
0:      (robot_attach_carrier jumper carrier2 central_warehouse)        [3]
0:      (robot_attach_carrier walker carrier3 central_warehouse)        [3]
3.001:  (robot_load_box drone carrier1 box1 central_warehouse two one)  [2]
3.001:  (robot_load_box jumper carrier2 box2 central_warehouse two one) [2]
3.001:  (robot_load_box walker carrier3 box3 central_warehouse two one) [2]
5.002:  (pick_up_content_at_cw drone carrier1 box1 valve)       [1]
5.002:  (pick_up_content_at_cw jumper carrier2 box2 tool)       [1]
5.002:  (pick_up_content_at_cw walker carrier3 box3 tool)       [1]
6.003:  (move_flying drone central_warehouse loc5)      [70]
6.003:  (move_jumping jumper central_warehouse loc1 loc2)       [30]
6.003:  (move_walking walker central_warehouse loc1)    [20]
26.004: (put_down_content walker carrier3 box3 loc1 tool wor1)  [1]
27.005: (move_walking walker loc1 central_warehouse)    [20]
36.004: (move_jumping jumper loc2 loc3 loc4)    [30]
47.006: (pick_up_content_at_cw walker carrier3 box3 valve)      [1]
48.007: (move_walking walker central_warehouse loc1)    [20]
66.005: (put_down_content jumper carrier2 box2 loc4 tool wor4)  [1]
68.008: (move_walking walker loc1 loc2) [20]
76.004: (put_down_content drone carrier1 box1 loc5 valve wor5)  [1]
88.009: (put_down_content walker carrier3 box3 loc2 valve wor2) [1]
```

Fig. 2. Solution found by PlanSys2 for problem instance 2.

```
root@docker-desktop:/home/package# ros2 run plansys2_terminal plansys2_terminal
[INFO] [1722783571.033807670] [terminal]: No problem file specified.
ROS2 Planning System console. Type "quit" to finish
> source launch/problem_3
> get plan
Plan not found
```

Fig. 3. Solution found by PlanSys2 for problem instance 3.

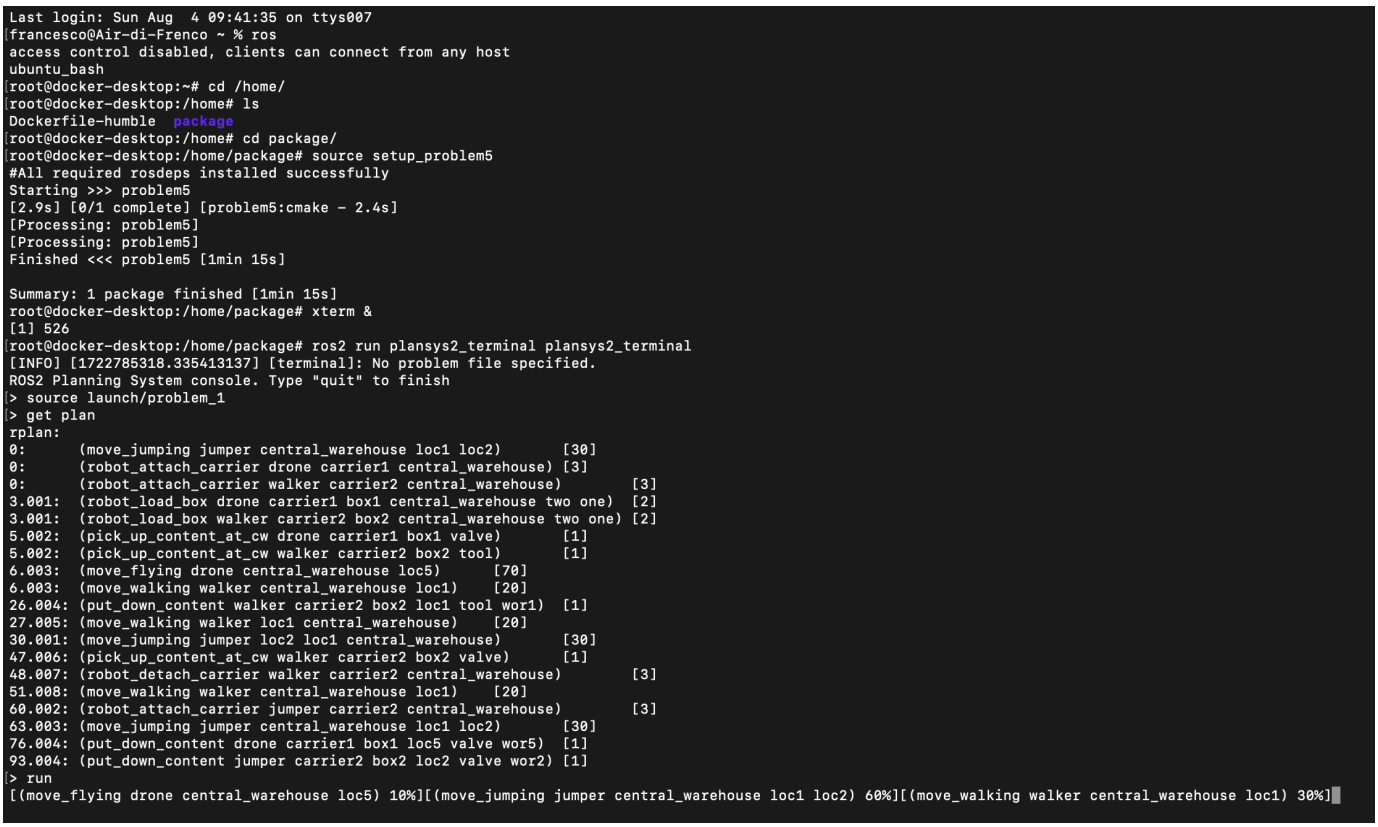

Fig. 4. Screenshot showing the build process of PlanSys2, the plan solution found for problem instance 1 and its execution, which is still in progress.