Automated Planning: Theory and Practice
Report
- Warehouse Distribution Center -

Andrea Cristiano
Artificial Intelligence Systems
Trento, Italy
andrea.cristiano@studenti.unitn.it
229370

Emiliano Rossi
Artificial Intelligence Systems
Trento, Italy
emiliano.rossi@studenti.unitn.it
229708

Abstract

In the following report we are going to present our solution to the planning problems proposed in the assignment of the course *Automated Planning: Theory and Practices*. Through this report, we will present our solutions, implementation choices, and assumptions that have allowed us to achieve the proposed results. The assignment consists of five different problems, with increasing difficulty. The context is that of a warehouse in which a robot must collect and distribute the requested items to various workstations. Starting from the simplest scenario, where the robot can transport only one object at a time, we will gradually increase the number of transportable items, add the temporal component and finally perform the entire task within PlanSys2, one of the comprehensive development environments in this field. The project source code is available on GitHub[1].

1 Introduction

In the context of *Artificial Intelligence*, automated planning[2] represents the process of developing algorithms and techniques that enable machines to generate plans or sequences of actions with the aim of achieving specific goals. Planning involves reasoning about a series of alternatives, defined as actions, that are executed in an environment in order to move from an initial state to a desired goal state.

This discipline has found application in various domains, such as robotics, scheduling, resource allocation, and process optimization.

In general, planners can be classified based on their approach:

- classic planning: in which we assume perfect knowledge of the environment;
- probabilistic planning: which accounts for uncertainty in actions or outcomes.

In this project we will focus on the classic planning approach, relying on *PDDL* (Planning Domain Definition Language) and *HDDL* (Hierarchical Domain Description Language) to model the planning problems, and on some of the most common planners to solve them.

The project scenario is inspired by industrial manufacturing services, involving several workstations located at known positions. To execute the assigned tasks, these workstations require specific items to be delivered to their locations. The scenario includes one particular location, called warehouse, where all the required items are initially held, along with at least one robotic agent capable of moving between locations to deliver the requested items. The objective of the planning systems is to coordinate the actions of the robotic agents to ensure that the workstations receive the necessary supplies.

The project is divided into five different problems, each characterized by a different level of complexity. They can be summed up as follows:

1. **Problem 1**: There is just one robot that can transport only one item at a time, from the warehouse to the workstations.
2. **Problem 2**: The robot must use a carrier to transport multiple items at a time. The number of items that can be transported at the same time depends on the carrier's capacity. There can be more than one robot and more than one carrier.
3. **Problem 3**: It is the same as Problem 2, but this time it must be modeled using a hierarchical task network (HTN).
4. **Problem 4**: It is the same as Problem 2, but this time it must be modeled using temporal planning.
5. **Problem 5**: It is the same as Problem 4, and it must be implemented in PlanSys2.

The report is organized as follows: in Section 2, you can find a description of all the elements that make up the project in general terms, providing an overview of the five problems that constitute the project. In Section 3, we analyze the characteristics of each individual problem: the assumptions made, the implementation choices, the tools used, and the results obtained.

Section 4 is dedicated to conclusions and our reflections. Finally, Section 5 provides a brief description of the project's structure and the files contained in the delivered folder.

2 Understanding the problem

The description of the project provided in the previous section is quite general and does not offer any details about the domain of the problems. In this section, we will provide additional elements to better understand the context in which the problems are framed.

2.1 General design of the environment

What we are going to describe in this section is part of the specific requirements outlined in the assignment and applies generally to all the problems.

- **Locations**: the environment is composed of a set of locations. There is a "road map" that describes the connections between the locations. Robots can move from one location to another only if they are adjacent.

- **Workstations**: there is a set of workstations, each located in a specific location, with the exception of the warehouse, which does not have any workstations. One location can contain more than one workstation. Each workstation may or may not require one or more specific item to be delivered to its location. To deliver an item to a workstation, the item must be in the same location as the workstation.

- **Boxes**: initially located in the warehouse, each box can contain any kind of item, but only one item at a time. A box can be loaded and unloaded by a robot. Loading can be done only if they are in the same location, while unloading can be done in any location.

- **Supplies**: they represent the items that the workstations require. Initially located in the warehouse, there can be different types of supplies (for example: bolts, tools, valves, etc.). To be delivered to a workstation, a supply must be placed in a box. A supply can be loaded and unloaded by a robot. Loading can be done only if they are in the same location, while unloading can be done in any location.

- **Robotic agents**: commonly called robots, they are the agents who will be responsible for delivering the items to the workstations. They can fill and empty boxes, load and unload boxes, move between locations and deliver the supplies to the workstations. To perform these actions, they must be in the same location as the object they want to interact with. At the beginning they are located in the warehouse. In the first problem they can transport only one item at a time, while, from the second problem on, they can use a carrier to transport multiple items at a time.

- **Carriers**: introduced in Problem 2, they are special objects that can be used by the robots to transport multiple items at a time. However, they have a limited capacity, and because of that it is important to keep track of the loaded number of items. As for the other objects, they are initially located in the warehouse. A robot cannot return to the warehouse until all the loaded items have been delivered to the workstations. A carrier can be attached to a robot only if they are in the same location.

2.2 Design choices

Certain aspects require further exploration to better define the context in which the problems are framed. At this point, we want to introduce the design choices made for the modeling and resolution of the problems.

- **Locations**: the environment is composed of four different locations: `warehouse`, `location1`, `location2` and `location3`. Warehouse is where all the objects are initially located, while location2 can be thought as a transit point, from which it is possible to reach all the other locations.

- **Workstations**: there are three workstations. `Workstation1` is located in `location1`, `workstation2` and `workstation3` are located in `location3`. No workstation is located in the warehouse and in location2. When an item is delivered to a workstation, it can no longer be used, since it is considered as consumed.

- **Boxes**: the number of the boxes varies depending on the problem. We start with `box1` in Problem 1, increasing the number in the following problems. When an item is delivered to a workstation, the box is emptied and can be used again to transport another item. Starting from Problem 2, the boxes can be loaded only into a carrier, and it can happen only if the carrier is already attached to a robot.

- **Supplies**: there are six supplies: two valves, `valve1` and `valve2`, two bolts, `bolt1` and `bolt2`, and two tools, `tool1` and `tool2`. An item can be loaded into a box only if the box is held by a robot, or if it is loaded into a carrier.

- **Robotic agents**: the number of robots varies from one to two, depending on the implementation of the problem. They are identified as `amr1` and `drone1`. Starting from Problem 2, they can use a carrier to transport multiple items at a time and cannot directly deliver an item. Because of that, in order to move between locations, robots must be attached to a carrier.

- **Carriers**: the number of carriers varies with the number of robots. If only one robot is present, there is only one carrier, `carrier1`, otherwise there will be also the second one `carrier2`. Not all the carriers are compatible with all the robots, it depends on their capacity. For example, a drone is considered compatible with a carrier defined as ”small”, which means that its capacity is less than or equal to 2. On the other hand, an amr is considered compatible with a carrier defined as ”big”, which means that its capability is greater than or equal to 3.

All the objects are defined in a way that allows us to introduce new ones without any particular effort. For example, if we want to add a new item that must be delivered to the workstations, we just need to add it in the problem file and define it as `supply`. Actions defined in the domain file will automatically apply to it. A graphical representation of the objects and their relationships is shown in Figure 1.

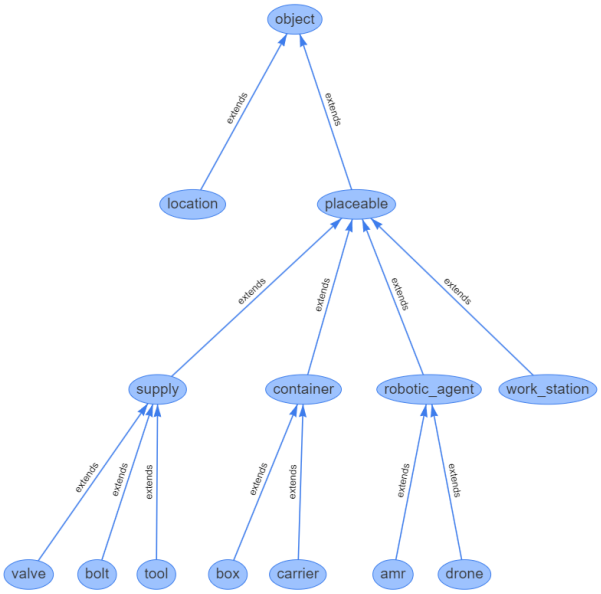

Figure 1: *Types graph*

With small variations depending on the problem, the structure of the domain file is the same for all the problems. The same can be said for the instantiating of the objects. An example of the instantiated objects is shown in Figure 2.

Both Figure 1 and Figure 2 are realized on the Problem 4 data and have been created using the *PDDL* add-on for VS Code[3].

Figure 3 provides a general graphical representation of the environment, with the objects and their initial locations.

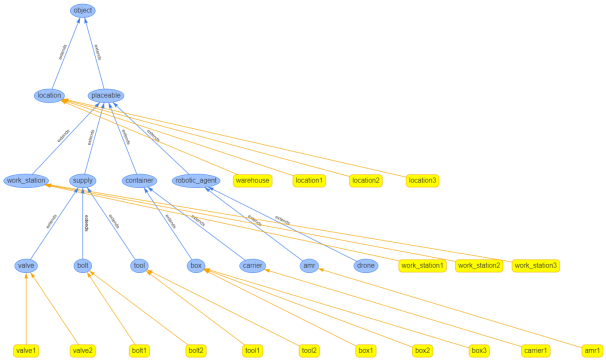

Figure 2: *Objects instantiating graph*

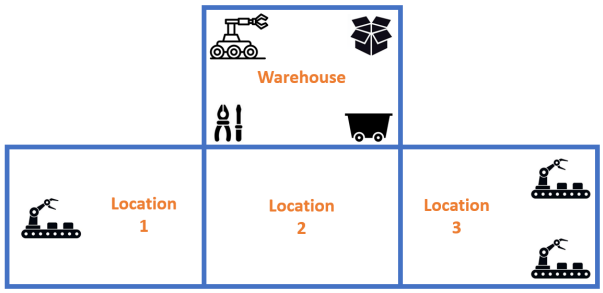

Figure 3: *Initial State*

The designed environment may seem quite simple and small. However, it is important to keep in mind that the number of states that need to be evaluated increases with the number of objects and possible actions. During the development of the project, we tried different environment configurations, such as varying the number of objects, locations, and actions. Given the available computational resources, what we are presenting is the best compromise we have found between complexity and the time required to solve the problems.

In terms of the planner used, the goals may vary slightly from one problem to another. On the one hand this is due to what we already mentioned in the paragraph above. Adding one more state to the goal means increasing the complexity to the point where the time required to solve the problem becomes too long. Different planners need different time to solve the same problem. Because of that, we increased, or decreased, the complexity of the goal accordingly with the planner we were using. At least, this is true for those planner that apply any kind of optimization strategy. Greedy planners, like *LAMA-First*, are not affected by this problem. More or less they always required the same time to find the solutions, but in most cases those were always the same, and completely unsatisfactory.

On the other hand, we were asked to model the problem in a way that the workstations don't specifically care about the content they receive. Although we intended to rely on `existential preconditions`[4] to meet this requirement, it was not possible as some of the planners used did not support it. To overcome this issue, we attempted to model the desired effect without explicitly using the existential precondition. One approach was to introduce additional predicates and actions in the *PDDL* domain to represent the conditions that the existential precondition would have captured. The advantage of this solution was achieving the desired behavior; however, the disadvantage is that such action is specific to only one type of item. If we aimed to achieve the same effect for all items, we would need to create as many actions as there are types derived from the supply type.

3 Problems

3.1 Problem 1

The first problem consists in the simple handling of a robot carrying different objects to diverse workstations located at different positions. From a general standpoint, our approach involved defining a robot that's able to pick up a box, fill it with a supply in the warehouse and then move to the corresponding location of a workstation to deliver the supply.

3.1.1 Domain

To better comprehend the full extent of our solution, firstly, we look at how we defined the environment. We defined four different locations, each of them of type `location`, that are distributed as shown in Figure 3. When two locations are adjacent, a robot can move from one to the other. That allows to create a graph map of adjacent areas. One special location is the *warehouse*, which is defined as a constant in *PDDL*, and it can be directly used in the actions. Since the `location` type expands its extent not to a single spot but more to a general area, workstations are defined as self standing objects `workstation` and different instances can be found in the same location.

Further important elements of our problem are the supplies and how we handle them: our robot needs to fill a box with a supply, in order to be able to transport it. In order to avoid creating specific actions for each specific kind of supply, we defined a generic type `supply` from which `bolt`, `tool`, `valve` derive, as shown in Figure 1. By doing so, it was sufficient to create the following actions to manage a wide variety of different objects:

- *fill_box*: to fill an empty box with a supply.
- *empty_box*: to remove a supply contained in a box.
- *deliver_supply*: to deliver a supply to a workstation and making the box void.

Last but not least, the robotic agent. Defined as `robotic_agent`, it can perform the following actions:

- *move_robot*: to move a robot from two adjacent locations.
- *load_robot*: to make a robot hold a box, it can be performed if the robot is not holding anything.
- *empty_robot*: to put down a pre-loaded box.

In order to be loaded, the boxes need to be in the same location as the robots. Once a box is held, it does not have any location, since its location is inherited by the location of the robot. Because at the beginning of the planning process every supply and box are located into the warehouse, the loading and filling actions can be performed only there. Conversely, the emptying actions (`empty_box` and `empty_robot`) can be done in any location. Once a supply has been delivered, the robot can either leave the box there or go back at the warehouse and reuse it.

3.1.2 Problem

We run the problem with two planners: **LAMA**[5] and **LAMA-First**. *LAMA* iteratively improves the solution, by helping itself with the previous found. Conversely, *LAMA-First* simply stops at the first solution found, using a greedy approach. Both planners found the optimal solutions: *LAMA-First*, being a greedy algorithm, does not guarantee an optimal solution. However, due to the simplicity of the problem it is able to achieve that solution. This stems from limited branching on the actions that can be performed to achieve the goal. There is no complex planning involved since the robot transports one resource at a time, picking up a box, filling it, and then delivering it.

Since the project is characterized by a series of problems of incremental difficulty, yet sharing various common aspects, in the following sections we will primarily focus on the differences introduced each time for solving the problem at hand.

3.1.3 Results

The following images, Figure 4 and Figure 5, display the obtained plans for *LAMA-First* and *LAMA*, respectively. As previously mentioned, due to the simplicity of the problem, the obtained plans are identical.

3.2 Problem 2 - Fluent approach

In the second problem, the general complexity increases by adding a new object `carrier`, declared as a new independent type. Boxes can now be moved to different locations if loaded into a carrier, each of which has a limited capacity. We also characterize the carriers based on their transportation capabilities.

We decided to address this problem by using two different approaches, due to the different planners

```
load_robot robot1 box1 warehouse (1)
fill_box robot1 box1 tool1 (1)
move_robot robot1 warehouse location2 (1)
move_robot robot1 location2 location1 (1)
deliver_supply robot1 box1 tool1 work_station1 location1 (1)
move_robot robot1 location1 location2 (1)
move_robot robot1 location2 warehouse (1)
fill_box robot1 box1 bolt2 (1)
move_robot robot1 warehouse location2 (1)
move_robot robot1 location2 location3 (1)
deliver_supply robot1 box1 bolt2 work_station3 location3 (1)
move_robot robot1 location3 location2 (1)
move_robot robot1 location2 warehouse (1)
fill_box robot1 box1 tool2 (1)
move_robot robot1 warehouse location2 (1)
move_robot robot1 location2 location3 (1)
deliver_supply robot1 box1 tool2 work_station3 location3 (1)
```

Figure 4: *LAMA-First solution*

```
load_robot robot1 box1 warehouse (1)
fill_box robot1 box1 tool1 (1)
move_robot robot1 warehouse location2 (1)
move_robot robot1 location2 location1 (1)
deliver_supply robot1 box1 tool1 work_station1 location1 (1)
move_robot robot1 location1 location2 (1)
move_robot robot1 location2 warehouse (1)
fill_box robot1 box1 bolt2 (1)
move_robot robot1 warehouse location2 (1)
move_robot robot1 location2 location3 (1)
deliver_supply robot1 box1 bolt2 work_station3 location3 (1)
move_robot robot1 location3 location2 (1)
move_robot robot1 location2 warehouse (1)
fill_box robot1 box1 tool2 (1)
move_robot robot1 warehouse location2 (1)
move_robot robot1 location2 location3 (1)
deliver_supply robot1 box1 tool2 work_station3 location3 (1)
```

Figure 5: *LAMA solution*

capabilities. In the first solution, which will be the subject of the 3.3 section, we tried to maintain a more classic approach, while in the second one, which will be the subject of this section, we used a planner **Metric-FF**[6] which allows us to use numeric values within the problem.

In this scenario, the robot must be attached to a carrier in order to deliver the supplies. The robot can attach or detach from a carrier using newly added actions:

- *attach_carrier*.
- *detach_carrier*.

Once the robot is attached to a carrier, it can relay on the actions explained in the previous section to deliver the supply. However, now the boxes cannot be loaded on the robot, but they must be put on the carrier. In order to further simplify preconditions and effects, we extended the previously explained non-location concept, to the carriers themselves: a carrier inherits its position from the robot it is attached to.

3.2.1 Domain

Each carrier is characterised by the following aspects:

- *max_capacity*: how many boxes can be carried at the same time.
- *loaded_volume*: how many boxes it is carrying.
- *carrying_requirements*: the minimum *power* needed by a robot to move the carrier.

On the other, the robots are extended with the `carrying_capability`, which represents the robots *power*.

With the presence of the carriers some actions changed to adapt to the new settings, while some others were introduced. Firstly, we modified the preconditions of each action, not allowing them to be performed if a robot has no carrier attached. Secondly, we shifted the `loaded` robot status, which indicates that a robot is holding a specific box, to the `contained` status, which indicates that a specific box is placed into the carrier. Now, in order to carry two supplies to a specific workstation, the robot can use a carrier that can load two or more boxes, fill them with the supplies, and deliver them. Thirdly, we impose that the robot cannot return to the warehouse until the carrier is empty.

We chose this setting for its simplicity. Different domains might have more refined actions; for example, we could model the loading of supplies onto a carrier by having the robot pick up the box, fill it, and then place it in an available carrier. However, we found this approach, while technically correct, to be impractical mainly due to algorithm limitations: breaking down the actions leads to the creation of intermediate states, which inevitably expands the search space, thus unnecessarily prolonging the search time.

3.2.2 Problem

The goal remains mostly unchanged. We employ various algorithms with different levels of completeness to compare their results and time requirements.

In this problem the optimization condition is also introduced: we are not only looking for a solution, but we aim to find the best one. With the fluent approach, we can achieve that by introducing a cost for each action: in an optimal setting, the robot should fill its carrier in such a way as to reduce the number of times it is forced to return to the warehouse for a new load. To assign costs to actions, we have decided to assign them empirically, making the cost dependent on the utility of an action. A very useful action (i.e. `deliver_supply`) should not be very costly, while a less useful action (i.e. `empty_box`) should be.

Solution to this problem are proposed in two different files: `problem_1` and `problem_2`. In the first one, we have the same condition as in problem 1, while in the second one, we introduce an additional robot and one more requirement in the goal.

3.2.3 Results

The `problem_1` requires to deliver three supplies, two of which are in the same location. The carrier is able to carry all of them.

The planner allows us to use three different algorithms that support optimisation:

- *A**.
- *A*epsilon*.
- *Enforced Hill climbing (EHC)*.

We know that if the heuristic function is admissible in `A*`, it is guaranteed to return the least-cost path from the start to the goal. To prevent an overshooting heuristic, we customized the weight to be as low as possible. We found that, in order to maintain an admissible heuristic, a standard weight (w) of 5 is sufficient. Specifically, we observed that the number of visited states changes with the weight: with $w = 3$, we have over 7000+ visited states, and with $w > 4$, it stabilizes around 1500.

The obtained solution is visible in Figure 6.

```
step    0: ATTACH_CARRIER CARRIER1 AMR1 WAREHOUSE
        1: LOAD_CARRIER AMR1 CARRIER1 BOX1 WAREHOUSE
        2: LOAD_CARRIER AMR1 CARRIER1 BOX2 WAREHOUSE
        3: FILL_BOX AMR1 CARRIER1 BOX2 VALVE2
        4: FILL_BOX AMR1 CARRIER1 BOX1 BOLT2
        5: LOAD_CARRIER AMR1 CARRIER1 BOX3 WAREHOUSE
        6: FILL_BOX AMR1 CARRIER1 BOX3 BOLT1
        7: MOVE_ROBOT AMR1 CARRIER1 WAREHOUSE LOCATION2
        8: MOVE_ROBOT AMR1 CARRIER1 LOCATION2 LOCATION1
        9: DELIVER_SUPPLY AMR1 CARRIER1 BOX2 VALVE2 WORK_STATION1 LOCATION1
       10: MOVE_ROBOT AMR1 CARRIER1 LOCATION1 LOCATION2
       11: MOVE_ROBOT AMR1 CARRIER1 LOCATION2 LOCATION3
       12: DELIVER_SUPPLY AMR1 CARRIER1 BOX1 BOLT2 WORK_STATION2 LOCATION3
       13: DELIVER_SUPPLY AMR1 CARRIER1 BOX3 BOLT1 WORK_STATION3 LOCATION3
       14: REACH-GOAL
plan cost: 9.500000
```

Figure 6: *Obtained solution for first case with A*, total cost 9.5*

The `A*epsilon` algorithm speeds up the search at the cost of losing the guarantee to find the optimal solution. Results can be improved by tweaking the weights once again. However, due to the randomness of the epsilon parameters, decreasing the weight does not always lead to an improvement. The best result we managed to achieve was with a weight of 3 (with 4666 states visited, 3000 less than `A*`), even though it is sub optimal.

The obtained solution is visible in figure 7.

The `Enforced Hill climbing` algorithm is the fastest. It returns a sub optimal plan visiting 214 states. When the number of actors in the problem increases too much it's the only one able to give a result in feasible time.

```
step    0: ATTACH_CARRIER CARRIER1 AMR1 WAREHOUSE
        1: LOAD_CARRIER AMR1 CARRIER1 BOX1 WAREHOUSE
        2: FILL_BOX AMR1 CARRIER1 BOX1 VALVE2
        3: LOAD_CARRIER AMR1 CARRIER1 BOX2 WAREHOUSE
        4: FILL_BOX AMR1 CARRIER1 BOX2 BOLT2
        5: MOVE_ROBOT AMR1 CARRIER1 WAREHOUSE LOCATION2
        6: MOVE_ROBOT AMR1 CARRIER1 LOCATION2 LOCATION1
        7: DELIVER_SUPPLY AMR1 CARRIER1 BOX1 VALVE2 WORK_STATION1 LOCATION1
        8: EMPTY_CARRIER AMR1 CARRIER1 BOX1 LOCATION1
        9: MOVE_ROBOT AMR1 CARRIER1 LOCATION1 LOCATION2
       10: MOVE_ROBOT AMR1 CARRIER1 LOCATION2 LOCATION3
       11: DELIVER_SUPPLY AMR1 CARRIER1 BOX2 BOLT2 WORK_STATION2 LOCATION3
       12: MOVE_ROBOT AMR1 CARRIER1 LOCATION3 LOCATION2
       13: EMPTY_CARRIER AMR1 CARRIER1 BOX2 LOCATION2
       14: MOVE_ROBOT AMR1 CARRIER1 LOCATION2 WAREHOUSE
       15: LOAD_CARRIER AMR1 CARRIER1 BOX3 WAREHOUSE
       16: FILL_BOX AMR1 CARRIER1 BOX3 BOLT1
       17: MOVE_ROBOT AMR1 CARRIER1 WAREHOUSE LOCATION2
       18: MOVE_ROBOT AMR1 CARRIER1 LOCATION2 LOCATION3
       19: DELIVER_SUPPLY AMR1 CARRIER1 BOX3 BOLT1 WORK_STATION3 LOCATION3
       20: REACH-GOAL
plan cost: 19.500000
```

Figure 7: *Obtained solution for first case with A*epsilon with w = 3, total cost 19.5*

The obtained solution is visible in Figure 8.

```
step    0: ATTACH_CARRIER CARRIER1 AMR1 WAREHOUSE
        1: LOAD_CARRIER AMR1 CARRIER1 BOX1 WAREHOUSE
        2: FILL_BOX AMR1 CARRIER1 BOX1 VALVE2
        3: MOVE_ROBOT AMR1 CARRIER1 WAREHOUSE LOCATION2
        4: MOVE_ROBOT AMR1 CARRIER1 LOCATION2 LOCATION1
        5: DELIVER_SUPPLY AMR1 CARRIER1 BOX1 VALVE2 WORK_STATION1 LOCATION1
        6: MOVE_ROBOT AMR1 CARRIER1 LOCATION1 LOCATION2
        7: EMPTY_CARRIER AMR1 CARRIER1 BOX1 LOCATION2
        8: MOVE_ROBOT AMR1 CARRIER1 LOCATION2 WAREHOUSE
        9: LOAD_CARRIER AMR1 CARRIER1 BOX2 WAREHOUSE
       10: FILL_BOX AMR1 CARRIER1 BOX2 BOLT2
       11: MOVE_ROBOT AMR1 CARRIER1 WAREHOUSE LOCATION2
       12: MOVE_ROBOT AMR1 CARRIER1 LOCATION2 LOCATION3
       13: DELIVER_SUPPLY AMR1 CARRIER1 BOX2 BOLT2 WORK_STATION3 LOCATION3
       14: MOVE_ROBOT AMR1 CARRIER1 LOCATION3 LOCATION2
       15: EMPTY_CARRIER AMR1 CARRIER1 BOX2 LOCATION2
       16: MOVE_ROBOT AMR1 CARRIER1 LOCATION2 WAREHOUSE
       17: LOAD_CARRIER AMR1 CARRIER1 BOX3 WAREHOUSE
       18: FILL_BOX AMR1 CARRIER1 BOX3 BOLT1
       19: MOVE_ROBOT AMR1 CARRIER1 WAREHOUSE LOCATION2
       20: MOVE_ROBOT AMR1 CARRIER1 LOCATION2 LOCATION3
       21: DELIVER_SUPPLY AMR1 CARRIER1 BOX3 BOLT1 WORK_STATION2 LOCATION3
       22: REACH-GOAL
plan cost: 21.500000
```

Figure 8: *Obtained solution for first case with EHC, total cost 21.5*

In `problem_2` the number of deliveries is increased to four, two for each location, and the available robots doubled. In this scenario the same pattern repeats: `A*` visits more states, thus requiring more time. At the price of losing the best solution, we can decrease the exploration with the other algorithms.

We show in Figure 9 the solution obtained with `A*` for the second case.

The main difference between the `A*` algorithm and `EHC` is the level of greed, with `A*epsilon` in the middle. While `A*` takes more time but explores more and is rewarded with the optimal solution, the intrinsic greedy policy of `EHC` blocks it from considering that loading more than one box in the carrier could lead to better results.

```
step    0: ATTACH_CARRIER CARRIER1 AMR1 WAREHOUSE
        1: LOAD_CARRIER AMR1 CARRIER1 BOX1 WAREHOUSE
        2: LOAD_CARRIER AMR1 CARRIER1 BOX2 WAREHOUSE
        3: FILL_BOX AMR1 CARRIER1 BOX2 TOOL2
        4: LOAD_CARRIER AMR1 CARRIER1 BOX3 WAREHOUSE
        5: FILL_BOX AMR1 CARRIER1 BOX3 VALVE2
        6: FILL_BOX AMR1 CARRIER1 BOX1 BOLT2
        7: MOVE_ROBOT AMR1 CARRIER1 WAREHOUSE LOCATION2
        8: ATTACH_CARRIER CARRIER2 DRONE1 WAREHOUSE
        9: LOAD_CARRIER DRONE1 CARRIER2 BOX4 WAREHOUSE
       10: MOVE_ROBOT AMR1 CARRIER1 LOCATION2 LOCATION3
       11: DELIVER_SUPPLY AMR1 CARRIER1 BOX1 BOLT2 WORK_STATION2 LOCATION3
       12: FILL_BOX DRONE1 CARRIER2 BOX4 VALVE1
       13: LOAD_CARRIER DRONE1 CARRIER2 BOX5 WAREHOUSE
       14: FILL_BOX DRONE1 CARRIER2 BOX5 TOOL1
       15: MOVE_ROBOT DRONE1 CARRIER2 WAREHOUSE LOCATION2
       16: MOVE_ROBOT DRONE1 CARRIER2 LOCATION2 LOCATION1
       17: DELIVER_SUPPLY DRONE1 CARRIER2 BOX4 VALVE1 WORK_STATION1 LOCATION1
       18: DELIVER_SUPPLY DRONE1 CARRIER2 BOX5 TOOL1 WORK_STATION1 LOCATION1
       19: DELIVER_SUPPLY AMR1 CARRIER1 BOX2 TOOL2 WORK_STATION3 LOCATION3
       20: REACH-GOAL
plan cost: 13.500000
```

Figure 9: *Obtained solution for second case with A*, total cost 13.5*

3.3 Problem 2 - Classic approach

In the classic approach we do not have the availability of numbers and their operations as in 3.2, thus we need to construct an algebra that resemble it.

3.3.1 Domain

To be able to resolve the problem, we defined a very simple algebra to represent the capacity of the carriers, the number of loaded boxes and the compatibility between the carriers and the robots. To increase the readability of the code, we used the `:derivatives` that allows us to define boolean functions within the preconditions of an action.

To keep track of the loaded element in the carriers, we introduced the type `quantity`. After having instantiated a set of quantities $\{n0, ..., n3\}$, we stated in their order in the problem file, with the predicates `increased_by_1` and `decreased_by_1` so achieving $n0 < n1 < ... < n3$.

By using the `contained` predicate we are able to know the amount of boxes contained into the carriers. Predicates `big_max_quantity` and `small_max_quantity` are used to define the maximum and minimum amount of boxes that the carriers can handle respectively. A carrier is full when the contained quantity is equal to the one assigned as maximum.

3.3.2 Problem

For this problem the scenario consists in having two robots delivering three supply. We keep using the *LAMA* and *LAMA-First* planners. For this last, as expected, the solution is produced in few instants but it lacks in clever moves such as collecting the supply needed in a location in a single carrier, while with time the improvement done by the *LAMA* algorithm manages to achieve the best solution.

3.3.3 Results

Figure 10 and Figure 11 show the results obtained with *LAMA-First* and *LAMA* respectively.

```
attach_carrier drone1 carrier2 warehouse (1)
load_carrier drone1 carrier2 box1 n0 n1 warehouse (1)
fill_box drone1 carrier2 box1 valve1 (1)
move_robot drone1 carrier2 warehouse location2 (1)
move_robot drone1 carrier2 location2 location1 (1)
deliver_supply drone1 carrier2 box1 valve1 work_station1 location1 (1)
move_robot drone1 carrier2 location1 location2 (1)
move_robot drone1 carrier2 location2 warehouse (1)
fill_box drone1 carrier2 box1 tool1 (1)
move_robot drone1 carrier2 warehouse location2 (1)
move_robot drone1 carrier2 location2 location3 (1)
deliver_supply drone1 carrier2 box1 tool1 work_station3 location3 (1)
move_robot drone1 carrier2 location3 location2 (1)
move_robot drone1 carrier2 location2 warehouse (1)
fill_box drone1 carrier2 box1 bolt2 (1)
move_robot drone1 carrier2 warehouse location2 (1)
move_robot drone1 carrier2 location2 location3 (1)
deliver_supply drone1 carrier2 box1 bolt2 work_station3 location3 (1)
```

Figure 10: *LAMA-First solution*

```
attach_carrier amr1 carrier1 warehouse (1)
load_carrier amr1 carrier1 box1 n0 n1 warehouse (1)
load_carrier amr1 carrier1 box2 n1 n2 warehouse (1)
load_carrier amr1 carrier1 box3 n2 n3 warehouse (1)
fill_box amr1 carrier1 box1 valve1 (1)
fill_box amr1 carrier1 box2 tool1 (1)
fill_box amr1 carrier1 box3 bolt2 (1)
move_robot amr1 carrier1 warehouse location2 (1)
move_robot amr1 carrier1 location2 location1 (1)
deliver_supply amr1 carrier1 box1 valve1 work_station1 location1 (1)
move_robot amr1 carrier1 location1 location2 (1)
move_robot amr1 carrier1 location2 location3 (1)
deliver_supply amr1 carrier1 box2 tool1 work_station3 location3 (1)
deliver_supply amr1 carrier1 box3 bolt2 work_station3 location3 (1)
```

Figure 11: *LAMA solution*

3.4 Problem 3

In Problem 3 we re-design the solution developed for Problem 2, using the hierarchical task network (HTN) approach. To find the solution, we use the **PANDA**[7] planner, which is able to handle HTNs.

It's important to note that the scenario remains the same as the one described in section 3.3. The only difference is the way in which the problem is modeled through the usage of the tasks and methods.

3.4.1 Domain

Hierarchical Task Network (HTN) planning technique is useful in domain where complex tasks can be broken down into a hierarchy of sub tasks. In turn, these sub tasks can be then broken down into a set of simpler tasks, called *primitive tasks*. This process can be done by introducing two new constructs: `:tasks` and `:methods` into the domain file.

Tasks represent actions at high level, with their own `:preconditions` and `:effects`. Methods represent the way in which tasks are broken down in sub tasks, defining how a task can be achieved. Each method is defined by the `:task` it achieves, and the `:subtasks` used to achieve the task. As for the tasks, methods can have `:preconditions`, `:effects` and `:ordering` to specify with which order the sub

tasks must be executed.

3.4.2 Problem

Looking at the problem file, we removed the `:goal` section as it became unnecessary since HTN introduces a new section called `:htn`.

A very interesting aspect of this approach is the ability to define an execution order for actions, allowing the most important ones to be executed first. However, in the proposed solution, we decided not to leverage this possibility, because in a sense this represents a constraint, leaving the choice of execution order to the planner.

3.4.3 Results

Figure 12 shows the results obtained with *PANDA*.

```
SOLUTION SEQUENCE
0: attach_carrier__DISJUNCT-0(amr1,carrier1,warehouse)
1: load_carrier__ANTECEDENT__DISJUNCT-1__DISJUNCT-1(amr1,carrier1,box4,n0,n1,warehouse)
2: load_carrier__ANTECEDENT__DISJUNCT-1__DISJUNCT-0(amr1,carrier1,box2,n1,n2,warehouse)
3: fill_box(amr1,carrier1,box2,bolt1,warehouse)
4: fill_box(amr1,carrier1,box4,valve1,warehouse)
5: load_carrier__CONSEQUENT___DISJUNCT-0(amr1,carrier1,box3,n2,n3,warehouse)
6: fill_box(amr1,carrier1,box3,tool1,warehouse)
7: no_op(amr1,carrier1,warehouse)
8: move_robot__DISJUNCT-0(amr1,carrier1,warehouse,location2)
9: move_robot__DISJUNCT-0(amr1,carrier1,location2,location1)
10: deliver_supply(amr1,carrier1,box4,valve1,work_station1,location1)
11: move_robot__DISJUNCT-0(amr1,carrier1,location1,location2)
12: move_robot__DISJUNCT-0(amr1,carrier1,location2,location3)
13: deliver_supply(amr1,carrier1,box2,bolt1,work_station3,location3)
14: deliver_supply(amr1,carrier1,box3,tool1,work_station3,location3)
```

Figure 12: *Problem 3 solution*

3.5 Problem 4

Problem 4 requires us to re-design our domain by using a temporal approach. We designed it in order to work with **OPTIC**[8] planner, which allows us to use numerical values.

3.5.1 Domain

The domain remain the same of Problem 2. Firstly, we convert the *actions* to *durative-actions*, by defining their duration. The main challenge was to organize the actions in such a way that there would not be unfeasible overlapping.

The first overlap arises when the same robot could perform two independent actions, with different preconditions, from the same state. To limit it, we introduced the *available* predicate that states the availability of the robot to perform an action. Whenever a robot performs an action, its available status becomes negative, going back to be positive only when the current task is completed.

The second conflicts is in the handling of supplies. It arises in those actions in which certain predicates are `true` in the precondition, and `false` in the effect. We settled that those predicates undergo changes at the start of the action. For example, a box *b*, in order to be loaded in a carrier, needs to be in a location *l*. Once on the carrier, the box has no location property. Hence, at the starting of the loading action we immediately set as `false` the predicate which located *b* in *l*. If the location of the box is not changed immediately, for a second robot the box would still be in the location, even if it is being loaded in a carrier, thus available to be picked.

The third interference happens when a predicate change its status as effect of an action. For example, if a robot were to unloading a box from its carrier and to set back the location of the box, an available robot could load it in its carrier even if the unloading is not finished yet. We solved this by imposing that, for this class of conflicts, the predicates are assigned at the end of the action.

3.5.2 Problem

We imposed a minimisation over the time execution of the plan. In the optimal plan, robots should perform actions in parallel and with carrier mostly fully loaded. The *OPTIC* planner follows the same principle of the *LAMA*: once found a first solution it refines with further computations.

The problem involves the delivery of four supplies, evenly distributed across two locations, with two available robots. While the initial plan partially parallelized the delivery of resources, subsequent computations attempted to optimize the plan. However, the algorithm still failed to find the optimal result within a feasible time frame. Some actions, such as moving out and returning to the warehouse, proved to be unnecessary.

3.5.3 Results

```
0.000: (attach_carrier carrier1 amr1 warehouse)  [4.000]
0.000: (attach_carrier carrier2 drone1 warehouse)  [4.000]
4.001: (load_carrier amr1 carrier1 box1 warehouse)  [2.000]
4.001: (load_carrier drone1 carrier2 box2 warehouse)  [2.000]
6.002: (fill_box amr1 carrier1 box1 valve1)  [1.000]
6.002: (fill_box drone1 carrier2 box2 bolt1)  [1.000]
7.003: (move_robot amr1 carrier1 warehouse location2)  [2.000]
7.003: (move_robot drone1 carrier2 warehouse location2)  [2.000]
9.004: (move_robot amr1 carrier1 location2 location3)  [2.000]
9.004: (move_robot drone1 carrier2 location2 location1)  [2.000]
11.005: (deliver_supply amr1 carrier1 box1 valve1 work_station3 location3)  [2.000]
11.005: (deliver_supply drone1 carrier2 box2 bolt1 work_station1 location1)  [2.000]
13.006: (move_robot amr1 carrier1 location3 location2)  [2.000]
13.006: (move_robot drone1 carrier2 location1 location2)  [2.000]
15.007: (move_robot amr1 carrier1 location2 warehouse)  [2.000]
15.007: (move_robot drone1 carrier2 location2 warehouse)  [2.000]
17.008: (fill_box amr1 carrier1 box1 tool1)  [1.000]
17.008: (fill_box drone1 carrier2 box2 tool2)  [1.000]
18.009: (move_robot amr1 carrier1 warehouse location2)  [2.000]
18.009: (move_robot drone1 carrier2 warehouse location2)  [2.000]
20.010: (move_robot amr1 carrier1 location2 location3)  [2.000]
20.010: (move_robot drone1 carrier2 location2 location1)  [2.000]
22.011: (deliver_tool amr1 carrier1 box1 tool1 work_station3 location3)  [2.000]
22.011: (deliver_tool drone1 carrier2 box2 tool2 work_station1 location1)  [2.000]
```

Figure 13: *Problem 4 solution*

3.6 Problem 5

The last problem involves implementing the solution developed for Problem 4 within **PlanSys2**[9]. *Plansys2* is a framework primarily designed for task and motion planning in robotics. It is built upon the **ROS**[10] (Robot Operating System). *ROS2* is an open-source framework for developing robotics applications. It provides a set of libraries, tools and algorithms for developing autonomous systems, in particular in the field of robotics.

The started solution has been extended to introduce all the elements for running it in the *PlanSys2* environment. For each action in the domain file, we implemented a corresponding *action node* file. Action nodes are responsible for simulating the actions and returning the results. Having no access to a physical robot, the only possibility was to simulate the actions. In section 5 it is possible to find a detailed list of the implemented action nodes and all the other elements for running the problem solution.

3.6.1 Results

PlanSys2 relies on **POPF**[11] as planner, and by using it we successfully generated valid plans. Given that the domain file is very similar to the one of the previous problem, we expected a similar solution as result. However, compared to what done for Problem 4, we decided to split the problem in two different cases, one with a single robot and one with two robots.

The following image shows the plan generated as solution for the problem with a single robot.

```
plan:
0:      (attach_carrier carrier1 amr1 warehouse)        [1]
1.001:  (load_carrier amr1 carrier1 box1 warehouse)     [2]
3.002:  (fill_box amr1 carrier1 box1 bolt1 warehouse)   [1]
4.003:  (load_carrier amr1 carrier1 box2 warehouse)     [2]
6.004:  (fill_box amr1 carrier1 box2 tool1 warehouse)   [1]
7.005:  (move_robot amr1 carrier1 warehouse location2)  [2]
9.006:  (move_robot amr1 carrier1 location2 location3)  [2]
11.007: (deliver_supply amr1 carrier1 box1 bolt1 work_station3 location3)       [2]
13.008: (deliver_tool amr1 carrier1 box2 tool1 work_station3 location3) [2]
15.009: (move_robot amr1 carrier1 location3 location2)  [2]
17.01:  (move_robot amr1 carrier1 location2 warehouse)  [2]
19.011: (fill_box amr1 carrier1 box1 tool2 warehouse)   [1]
20.012: (move_robot amr1 carrier1 warehouse location2)  [2]
22.013: (move_robot amr1 carrier1 location2 location1)  [2]
24.014: (deliver_tool amr1 carrier1 box1 tool2 work_station1 location1) [2]
```

Figure 14: *Problem 5 solution with one robot*

In this plan the robot delivers all the required items to one workstation and then the remaining one to the other.

The following image shows the plan generated as solution for the problem with two robots. One robot performs pretty well: it attaches to the selected carrier, loads boxes and supplies, and delivers them. The other robot instead simply wanders around for a while. All the elements are delivered by the first robot.

This plan is very similar to what was obtained for Problem 4, and as in that case the algorithm failed to find the optimal result. The robot that wanders around could have delivered the item in the same time as the other robot, but it did not.

```
plan:
0:      (attach_carrier carrier1 amr1 warehouse)        [1]
0:      (attach_carrier carrier2 drone1 warehouse)      [1]
1.001:  (move_robot amr1 carrier1 warehouse location2)  [2]
1.001:  (load_carrier drone1 carrier2 box1 warehouse)   [2]
3.002:  (fill_box drone1 carrier2 box1 bolt1 warehouse) [1]
3.002:  (move_robot amr1 carrier1 location2 location1)  [2]
4.003:  (load_carrier drone1 carrier2 box2 warehouse)   [2]
6.004:  (fill_box drone1 carrier2 box2 tool1 warehouse) [1]
7.005:  (load_carrier drone1 carrier2 box3 warehouse)   [2]
9.006:  (fill_box drone1 carrier2 box3 valve1 warehouse)        [1]
10.007: (move_robot drone1 carrier2 warehouse location2)        [2]
12.008: (move_robot drone1 carrier2 location2 location1)        [2]
14.009: (deliver_supply drone1 carrier2 box1 bolt1 work_station1 location1)     [2]
16.01:  (move_robot drone1 carrier2 location1 location2)        [2]
18.011: (move_robot drone1 carrier2 location2 location3)        [2]
20.012: (deliver_supply drone1 carrier2 box3 valve1 work_station3 location3)    [2]
22.013: (deliver_tool drone1 carrier2 box2 tool1 work_station3 location3)       [2]
24.014: (move_robot drone1 carrier2 location3 location2)        [2]
26.015: (move_robot drone1 carrier2 location2 warehouse)        [2]
28.016: (fill_box drone1 carrier2 box1 tool2 warehouse) [1]
29.017: (move_robot drone1 carrier2 warehouse location2)        [2]
31.018: (move_robot drone1 carrier2 location2 location1)        [2]
33.019: (deliver_tool drone1 carrier2 box1 tool2 work_station1 location1)       [2]
```

Figure 15: *Problem 5 solution with two robots*

4 Conclusions

In this report we have discussed the approaches used to solve the problem of delivering supplies to different workstations in an industrial environment. We have implemented the solutions using different planners and different approaches. We also have discussed the results obtained and the problems encountered.

The implementation of the HDDL was the most critical one. Compared to all the other proposed solution, we had to change our approach, relying on a bottom-up approach.

The second critical aspect was to install and configure the *PlanSys2* framework. The installation process was not straightforward and required a lot of time to be completed. At the end of the day we had to install the framework on a different operative system, because of some incompatibilities with the one we were using.

We'd like to conclude by stating that we have come to appreciate the complexity inherent in planning. In particular, we observed a variety of planners, each of which required its own tweaking and configuration to function effectively. Furthermore, we noticed how challenging it is to find the best solution, even in a relatively simple scenario, and how a solution highly depends on the designed domain. The simpler the domain the easier is for a planner to find a good plan. We believe that focusing on simplification, without losing information, is the key to being able to search for a good solution within a reasonable time.

4.1 Future Works

We believe that the project can serve as a starting point for more complex applications, perhaps by exploring new alternatives and different approaches, feasible for real-world scenarios. With the evolution of the field

of *Automated Planning*, there will increasingly be contexts in which it could find applications.

5 Archive organization

The project is divided into five different folders, each dedicated to a specific problem. Each of them containing primarily the following files:

- *domain.pddl*;
- *problem.pddl*;
- *Results.md*.

For each problem we used different planners, here below the complete list:

Problem 1	LAMA-First, LAMA
Problem 2	LAMA, Metric-FF
Problem 3	PANDA
Problem 4	OPTIC
Problem 5	POPF

To use the proposed solutions, it is necessary to install the planning tools and configure the development environment.

For solutions to Problems 1, 2, and 4, **Planutils**[12] tool was used, since it provides the user with a wide variety of planners that are easily installable and ready to use. In solving Problem 3, *PANDA* was used, a planner mainly developed in *Java* for which it is necessary to install the *Java Virtual Machine*. Finally, to address Problem 5, *PlanSys2* along with *ROS2* were required.

The project was developed and tested on *Ubuntu 20.04 LTS* and *Ubuntu 22.04 LTS*. However, with appropriate adjustments, it is possible to use the proposed solutions on other operating systems and/or planning systems.

For correct installation and configuration of the tools, please refer to the official documentation.

Inside the *Result.md* files of each problem, it is possible to find instructions for executing the proposed solutions.

References

[1] *Project Repository*. URL: https://github.com/andy295/Automated-Planning-Project.

[2] *Automated Planning*. URL: https://en.wikipedia.org/wiki/Automated_planning_and_scheduling.

[3] *PDDL VSCode add-on*. URL: https://marketplace.visualstudio.com/items?itemName=jan-dolejsi.pddl.

[4] *Existential Preconditions*. URL: https://planning.wiki/ref/pddl/requirements#existential-preconditions.

[5] Matthias Westphal Silvia Richter. "The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks". In: *Journal Of Artificial Intelligence Research* Volume 39 (2010), Pages 127–177. DOI: https://doi.org/10.1613/jair.2972.

[6] *Metric-FF*. URL: https://fai.cs.uni-saarland.de/hoffmann/metric-ff.html.

[7] *PANDA*. URL: https://www.uni-ulm.de/en/in/ki/research/software/panda/panda-planning-system/.

[8] *OPTIC*. URL: https://nms.kcl.ac.uk/planning/software/optic.html.

[9] *PlanSys2*. URL: https://github.com/PlanSys2/ros2_planning_system.

[10] *ROS2*. URL: https://docs.ros.org/en/humble/.

[11] *POPF*. URL: https://nms.kcl.ac.uk/planning/software/popf.html.

[12] *Planutils*. URL: https://github.com/AI-Planning/planutils.