Emergency Services Logistics Problems

Davide Modolo (#229297)[1], Francesco Laiti(#232070)[2]

Abstract

This report explains the development and implementation using PDDL/HDDL planners of five different scenarios involving injured people and delivery robots, with increasing complexity. We addressed and explained our solutions and the assumptions we made to solve the problems, following the instructions provided in the assignment of the course. Starting from a basic domain, we moved to a more complex scenario, then addressed the problem with hierarchical task networks, converted the domain using durative actions, and finally, implemented the last problem within PlanSys2. We successfully built and ran domains and problems to find a valid plan in every scenario. The source code is available at `github.com/laitifranz/emergency-logistics-ap`.

[1] *davide.modolo@studenti.unitn.it*
[2] *francesco.laiti@studenti.unitn.it*

1. Introduction

Automated planning is a branch of artificial intelligence that enables machines to decide a sequence of actions that will lead them from an initial state to a desired goal state. Using a planner, the algorithm takes a domain and a problem as input and produces a valid plan as output, if it exists.

The main goals of this assignment are to model planning problems using PDDL (Planning Domain Definition Language) and HDDL (Hierarchical Domain Description Language) and use state-of-the-art planners to solve them.

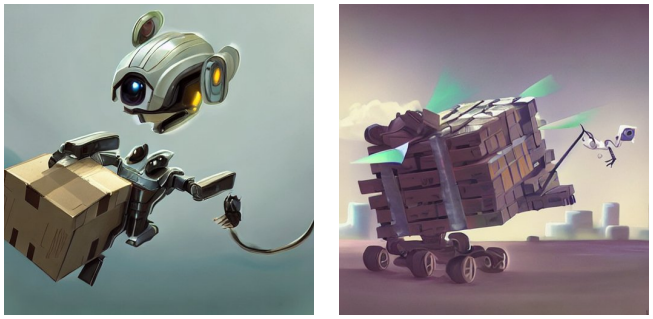
The considered general scenario is inspired by an emergency services logistics problem. The objective is to use robotic agents to deliver boxes containing emergency supplies to injured individuals in fixed positions. The five tasks, in brief, are:

1. *Problem 1*. One robotic agent, that can carry only one box, has to deliver the supplies;
2. *Problem 2*. Extension of Problem 1. The robotic agent can now attach itself to a carrier that can load up to four boxes;
3. *Problem 3*. Address Problem 2 using hierarchical task networks (HTN), introducing `:tasks` and `:methods`;
4. *Problem 4*. Convert the domain of Problem 2 to use durative actions;
5. *Problem 5*. Implement within `PlanSys2` the Problem 4 using "fake actions".

For the evaluation of our solutions, we created alternative versions of domain and problem files to address specific issues or to perform extra tests, in addition to the standard ones.

To run the code we adopted the planners suggested during the course. We worked exclusively on the Docker platform to avoid compatibility issues between different operating systems and have a reliable environment that can be easily reproduced and shared. The configuration for each problem has been provided in our repository.

The planners used in this project are: **planning.domains** using their API [1] in Python to get a plan; **LAMA/LAMA-first**, **OPTIC** and **TDF** using *planutils* [2] image; **PANDA** [1], downloaded from the official source [5]; and **PlanSys2** [6].



(a) 1^{st} task (one box per robot). **(b)** $\geq 2^{nd}$ tasks (one carrier per robot with many boxes).

Figure 1. How we imagine the robots for the different scenarios; images created with Stable Diffusion Online [7].

2. Understanding of the problem

As already explained in Section 1, the scenario is inspired by an emergency services logistics problem, where a number of persons, in specific locations, have been injured.

We were tasked to address this problem in five different scenarios. The second scenario is built on the first, the third and the fourth are built on the second and the fifth is built on the fourth.

The following assumptions have been made:

1. **Injured people** are stationary and located at specific positions. The location is known a priori and it can not be the depot;

[1] We needed the Docker image of Java Version 8 [3] as recommended by the authors [4].

2. **Each injured person** has or does not have some specific content. Therefore, each person might need all the supplies, while others just one or more specific supplies, or nothing;
3. **Multiple people** can be at the same location, and therefore it isn't sufficient to keep track of where a supply is in order to know which people have been given the required supplies;
4. **Supplies** are packed in boxes, represented as their contents. As suggested in the assignment, there are three kinds of supplies: food, medicine, and tools, all stored initially at the depot (still, new types of supplies can be added thanks to their encoding as sub-objects of the 'supply' object, as shown in Figure 2);
5. **Boxes** are initially placed at the depot and can contain different supplies. They can be filled with one supply at a time;
6. **Robotic agents** can fill, empty, pick up, and drop boxes and also move and deliver them (depending on the problem) where they are needed;
7. **Robotic agents** can move directly between any location because we assume that the locations graph is fully connected. Initially, any robotic agent is located at the depot;
8. **Carrier** is introduced in Problem 2 and enables robotic agents to load up at most 4 boxes and to move and unload them in different locations. The robotic agent does not have to return to the depot until after all boxes on the carrier have been delivered. In those problems, the robotic agent loses the ability to pick up a box for itself;
9. **Carrier capacity** is an important aspect to monitor, in order to prevent the robot from overloading the carrier. Initially, we assume it to be 0 and it can be increased up to 4 (indicating that the carrier can load up a maximum of four boxes).

The goal that we want to achieve is common to all the scenarios: deliver 2 different supplies to a specific person and 1 kind of supply to two people located in different known-a-priori locations. Even if we did separate tests with different goals to verify that our solution worked properly, we left the same goal for each problem for the final version of the project to better compare the different implementations.

As remarked in the assignment, injured persons are more concerned with the general category or type of supply rather than its specific instance (if two people need food, it doesn't matter which instance of the food object each of them receives).

3. Design choices

In this section, we discuss our implementation and the design choices to address the five scenarios. We describe step-by-step every task, each in its own sub-section. At the beginning of every domain code, we left comments that can help to guide the reader with our improvements or changes to the design of the solution dictated by unsupported features from the different planners tested and the different requirements from the assignment.

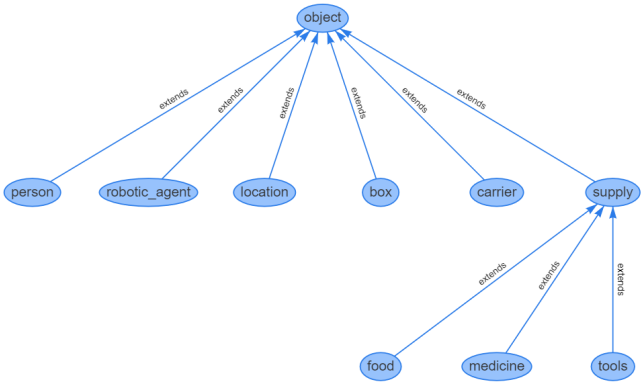


Figure 2. *Types* graph for Problem2 domain created using a PDDL add-on for VS Code [8].

General design choices In general, all the problems share a common initial state and the same goal. We declared the types of our problem in the domain file (Figure 2 gives an example of a visual illustration of our objects), and then we assigned the objects instances in the problem file.

We list the initial state and constraints in the problem code as follows (Figure 3 shows an example of the instantiated objects):

- **Locations** are four, the `depot` and the three spots `location1`, `location2`, `location3`. Initially, robot, supplies, boxes, and carrier are located at the depot, and no person is there;
- **Injured person** are three, `per1`, `per2`, `per3`, located in three different locations;
- **Boxes** are at most 4, and we declared them as `box<#>`. Boxes are initially empty and they are not loaded on the robot (for the first problem) or on the carrier (for the other problems). The boxes can be empty or can contain only one supply, so a box can be used to satisfy the needs of one injured person at a time. The boxes can be reused, but they need to be unloaded from the carrier to be filled with the next supply (that needs to be in the same location);
- **Supplies** are four: one medicine `med1`, two foods, `food1` and `food2`, and one tool `tools1`;
- **Robotic agent** is declared as `robot1` and initially is located at the depot. In Problem 1 the agent can transport one box, while in the other problems the agent relies on a carrier to transport the boxes;
- **Carrier** is one, `carrier1`, and it is available from Problem 2 onward. It is initially located at the depot and it is detached from the robot. We assume that the agent needs to attach the carrier before using it, and

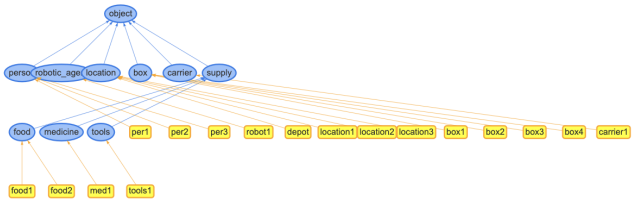


Figure 3. Objects graph for Problem 2 problem file, created using a PDDL add-on for VS Code [8].

it can be detached if needed. Initially, the carrier has no boxes loaded and its maximum capacity is 4. We also assumed that, in the problems with a carrier, the robotic agent loses its ability to carry a box on its own, requiring the use of the carrier even in a hypothetical scenario with only one box.

The goal that we set is the one schematized in a comprehensive (pseudo-code) way in Figure 4. Note that we structured the goal to accomplish the remark of the assignment, as described at the end of Section 2. We assume that the goal is ended when the delivery of supplies required by the injured person is completed.

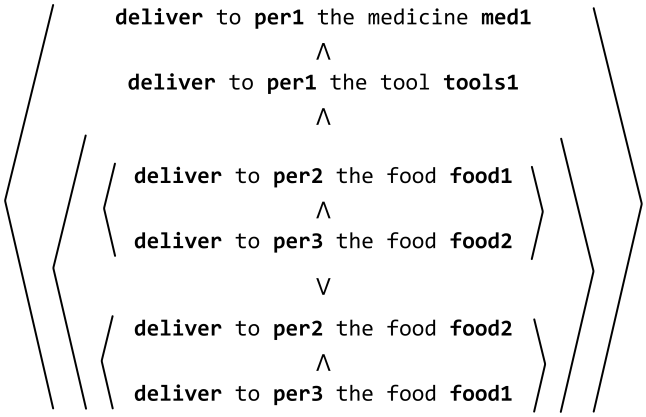


Figure 4. Our goal.

3.1 Problem 1

Problem 1 defines the basic setup of this project and it is used as the baseline for the next problems.

In this scenario a single robot is introduced, `robot1`, that can carry one single box at a time. The goal is to deliver each person the supplies they need.

In our problem, both `per2` and `per3` needed food, and `per1` needed tools and medicine. Since people don't care exactly which content they get, our goal includes the `or` condition for the food delivery. The `delivery` action, symbolizes the act of emptying a box to give the supply to the injured person.

3.1.1 Predicates & Actions

In this subsection, we provide a full list of the predicates and actions. We chose names that, we hope, are self-explanatory.

Predicates

- located_at (either robotic_agent box supply person)
- robot_has_box
- robot_has_no_box
- box_with_supply
- box_is_empty
- delivered

Actions

- move_robot
- take_box
- drop_box
- fill_box
- empty_box
- deliver

3.2 Problem 2

In Problem 2, we take Problem 1 with some extensions, by changing how boxes are carried around. Now, instead of carrying one box at a time, a robotic agent can be attached to a carrier that can transport at most 4 boxes. For this reason, we keep track of which boxes are on the carrier and how many boxes are loaded, to avoid being overloaded. Additionally, the carrier needs to be initially at the depot and we assume that there are no boxes loaded.

To load a box on the carrier, the agent, the box and the carrier need to be in the same location.

To deliver a supply, the box containing it has to be unloaded first from the carrier.

We created three different versions of this problem:

- the first version is more compact but supported only by the API using Python. In this version of domain and problem, we use `:numeric-fluents` that allow us to write the maximum number of boxes that can lay on a carrier as a variable;
- the second one doesn't include `:numeric-fluents`, and it is used as a baseline for Problems 3 and 4, and it also allowed us to use the `LAMA` planner;
- the third version is only a modification of the problem file of the second version, where we declare only 3 boxes and the goal is still to have 4 deliveries. This helps us to show the ability and the possibility to use multiple times a single box with different supplies.

The first version, using `:numeric-fluents`, differs from the second version only by having the variable for the boxes and the required changes to the actions. We introduced the `:functions (num-boxes)` to store the number of boxes on a specific carrier. `num-boxes` is a variable that maintains a value throughout the duration of the plan.

In the second version, we set up separate predicates for the maximum number of boxes a carrier can hold, and adjusted the actions' code accordingly.

We now present the predicates and the actions implemented in the domain file of the second version described above.

3.2.1 Predicates & Actions

To avoid long lists, we only provide the extensions of predicates and actions inherited from Problem 1.

Predicates added

- box_on_carrier
- box_loaded
- carrier_has_no_robot
- robot_has_no_carrier
- robot_carrier_attached
- carrier_has_no_boxes
- carrier_has_one_box
- carrier_has_two_boxes
- carrier_has_three_boxes
- carrier_has_four_boxes

Actions added

- move_robot_with_carrier
- attach_carrier_to_robot
- detach_carrier_from_robot
- load_box_on_carrier
- unload_box_from_carrier

The operations and conditions of the carrier, such as loading, unloading and moving have been fixed to be up to 4 loading positions, as mentioned earlier. This allows us to describe carriers that can hold up to 4 boxes using these operations/conditions. To increase the capacity of the carrier, it is necessary to make some changes. However, this is easy because it is only necessary to add the predicates `carrier_has_<#>_boxes` up to the number of boxes that we want to use, and then modify the actions in the code accordingly.

3.3 Problem 3

In Problem 3, we address Problem 2 with hierarchical task networks (HTN) by breaking down complex tasks into simpler sub-tasks. As required by the HDDL language, we need to introduce `:tasks` and `:methods`. As suggested in the assignment, we keep the same actions as per the solution of Problem 2, the same initial conditions, and the same goal. For the goal, we introduce these goal tasks, sequentially ordered:

1. (task0 (T_deliver_supply per1 med1))
2. (task1 (T_deliver_supply per1 tools1))
3. (task2 (T_food_delivery_satisfied per2 per3 food1 food2))

We needed to redesign the goal from Problem 2 because `:disjunctive-preconditions` is not supported by `PANDA` planner. For this reason, we created a task `T_food_delivery_satisfied` to overcome the limitation.

We now present the tasks, methods, predicates, and actions implemented in the domain file.

3.3.1 Predicates, Tasks & Methods

Since the `either` keyword wasn't supported, we wrote the locations of every object as separate predicates. Any predicate or action already implemented in the previous problem is not present in the following lists (see Section 3.2).

Predicates added

- located_at_carrier
- located_at_robot
- located_at_box
- located_at_supply
- located_at_person

Primitive Tasks Primitive tasks, with their primitive methods, directly call an action.

- T_move_robot
- T_attach_carrier_to_robot
- T_detach_carrier_to_robot
- T_fill_box
- T_load_box_on_carrier
- T_unload_box_from_carrier
- T_deliver

It is interesting to notice that the task `T_move_robot` is declared once, and depending on the current state, it can either execute the method to move the robot with the carrier or without it.

Non-Primitive Tasks Non-primitive tasks are not linked to any action and they cannot be executed directly. They, instead, have to be divided into smaller tasks, which can be either simple or non-primitive tasks, which in turn will be divided.

- T_food_delivery_satisfied
- T_deliver_supply
- T_return_to_depot
- T_prepare_box

The first two tasks are the ones used in our goal. In particular, the `T_food_delivery_satisfied` sub-goal performs, with its methods, the `T_deliver_supply` twice to satisfy the delivery of food to each person independently from the food instance they receive, as illustrated in Figure 4.

Primitive Methods

- M_move_robot
- M_move_robot_with_carrier
- M_attach_carrier_to_robot
- M_detach_carrier_to_robot
- M_fill_box
- M_load_box_on_carrier
- M_unload_box_from_carrier
- M_deliver

Non-Primitive Methods

- M_food_delivery_satisfied_method1
- M_food_delivery_satisfied_method2
- M_deliver_supply_with_already_carrier
- M_deliver_supply_but_first_attach_carrier
- M_deliver_supply_by_loading_supply_from_depot
- M_return_to_depot
- M_prepare_box

The first two items are methods for the task `T_food_delivery_satisfied`, where we left an order for the execution of the sub-tasks. This is not mandatory but we observed that removing the order from these methods reduces the solution steps by two. It is interesting to notice how the plan provides a solution where the robot fills and loads two boxes at the same time to optimize the delivery task, whereas the solution where we left the ordering resulted in a sub-optimal plan with the robot filling and delivering one box at a time by going back-and-forth from the depot. The solution left on the repository is the ordered version, but it can be easily tested by commenting the `:ordering` section in the corresponding methods.

Actions no new actions have been added. All are inherited from Problem 2.

3.4 Problem 4

In Problem 4 we implemented the temporal domain by adding a duration to the actions of Problem 2. To achieve this, we enhanced all the actions by specifying a time duration and incorporating time constraints that must be met `at start`, `over all`, and `at end` of the actions. These constraints prevent simultaneous execution of actions that, based on our design, cannot be run in parallel.

We tried to be consistent with a real-world application, by assigning these durations to the actions:

- *move_robot* has duration 5;
- *move_robot_with_carrier* has duration 7, since carrying the weight should reduce the speed of the robotic agent;
- *attach_carrier_to_robot* & *detach_carrier_from_robot* have duration 2;
- *load_box_on_carrier* & *unload_box_from_carrier* have duration 3, since we thought that a box filled with a supply would be heavy;
- *fill_box* has duration 4, since our idea is that the robot has to physically search and take the required supply when inside the depot;
- *deliver* has duration 3, because it implies that the box is already off the carrier.

Differently from Problem 2, the planners we tested for this problem did not support the requirement `:disjunctive-preconditions`, as already addressed in Problem 3. Thus we had to redesign our goal. We had to rewrite the `OR` part of the goal by introducing dummy actions. We added two new durative actions:

- delivery_OR_refactored_possible_action1 to satisfy the goal of delivery `food1` to `per2` and `food2` to `per3`;
- delivery_OR_refactored_possible_action2 to satisfy the goal of delivery `food1` to `per3` and `food2` to `per2`;

In this way, we were able to add the goal to the problem correctly. Initially, we didn't see these actions in the plan since we set their duration to zero. Because we considered them as instant actions, we applied a very short duration to show them in the final plan.

Thanks to the support of the planners tested of `:numeric-fluents`, we were able to use `:functions` to monitor the number of boxes on the carrier and use `increase` or `descrease` in the effects. This simplified the writing of the domain.

Another problem that we faced was that `:negative-preconditions` was not supported by the `OPTIC` planner and, spoiler, neither by the planner in Problem 5. Therefore, to overcome this limitation, we added to the predicates of the problem `box_not_on_carrier`, in order to track if a specific box has been loaded on the carrier. We also properly added the new predicate to the initial state.

As a final test, we also made a problem version where the robotic agent has only 3 boxes available to see if the planner could find a valid plan, as already done in Section 3.2.

3.5 Problem 5

In Problem 5, we have expanded upon the framework set of Problem 4 to simulate the duration and execution of actions using associated C++ codes. For every action mentioned in Section 3.4, a specific C++ code has been developed. These codes function as "fake actions" and have a predefined duration.

It is important to note that these codes run sequentially, avoiding parallelization, and the planner (the default one is `POPF` [9]) doesn't support `:negative-preconditions`. That's why we relied on a PDDL version that does not have negative preconditions in `:condition` of `:durative-action`. Additional predicates defined in Section 3.4 are kept to overcome this limitation.

To run this task, it's required to have two different terminals of the same container opened. More details on the setup can be found in our repository.

4. Results

In this section, we discuss the results obtained by running the problems with different planners. For every problem, we obtained at least one valid plan. In our repository, we report, in a fashion style, the output of the planners in the README file, under its *Results* section.

4.1 Problem 1

The plans for the first problem have been generated using two planners: `LAMA` and `planning.domains` API. We decided to use `LAMA` and not `LAMA-first` because we wanted

to compute a complete search in the solution space, and because the problem is the simplest one, our machines can handle the search. In fact, we obtained the optimal solution in less than a minute.

Comparing the two solutions output from the two planners, we can observe that the length of the two plans is the same, and they differ only in steps 5 and 6, where the plan we got from LAMA moves the robot to the depot to take another box, while in the planning.domains plan, the robot just delivers the supplies using only one box. We remember to the reader that the robot has two boxes at disposition but can load only one box at a time on itself.

To close the considerations of this problem, we can confirm that, looking at the solution itself, the plans are optimal, also confirmed by the LAMA planner that terminates the search.

4.2 Problem 2

The plans have been generated using, again, LAMA and, with our Python script, the planning.domains API. We made two versions, one that uses :numeric-fluents, supported by the planning.domains API, and the other one without it. We tried also with LAMA-first, but for a more depth search, we adopted LAMA.

4.2.1 Without :numeric-fluents

In this case, the problem is more complex than before. We noticed that the planner took lots of time to explore the search space in search of the optimal solution.

LAMA gradually provides better plans when it finds them. After ten minutes, the planner stopped the search and returned the optimal solution. It is interesting to notice that the best plan has been found after some seconds, and no other better solutions were found.

Also here, after looking at the solution, we can say that the carrier is fully-exploited to move the boxes and deliver the supplies in the smartest way. The search space of this problem, w.r.t. the previous one, increased a lot.

Extra problem with 3 boxes available We applied in this problem the scenario where only 3 boxes were available, to see how the planner generates the plan. The plan has been generated using LAMA again, and surprisingly, it takes less than a minute to conclude the search and find the optimal solution. The plan is 3 steps longer than the previous case because it has to reuse some boxes to complete the delivery.

So, we can see that having only three boxes available forces the robotic agent to first deliver the three filled boxes and then take the carrier with one of the boxes to return to the depot in order to fill it up with the last needed supply (and deliver it).

4.2.2 With :numeric-fluents

With the introduction of :numeric-fluents, we need to run our code using the planning.domains API for Python. We would point out that the planner returns the solution in a couple of seconds, and thus we think that it returns the first admissible solution. Looking at the plan returned, we can see that it is a sub-optimal solution, longer than the output plans obtained above. It returns an inefficient solution, where the robotic agent moves one box at a time, even if it fills them all at once, and, after it unloads all the boxes in the right location, it needs to revisit the location again to deliver the supply.

4.3 Problem 3

For this problem, we generated the plan using the HTN planner PANDA.

Initially, we set an order for the sub-goals, to determine the sequence in which tasks should be performed. Our idea was to simulate the scenario where people with worse health conditions need higher priority (and some of them should be served before others), and people who need tools have less priority than people who need food and/or medicine.

Observing the resulting plan, we can see that the robot works with only one box at a time, resulting in a sub-optimal solution. With no additional settings, the planner returns only the first solution found.

After that, we tried to remove the ordering of the sub-goals for test purposes. Unfortunately, we were not able to test this case properly. Even after increasing the heap size of the JVM to 26GB (the maximum quantity of RAM we had available), the planner didn't find any solution even after an hour and a half and almost 26 million generated nodes.

We can say that introducing a hierarchy of goals is very important to guide the planner toward finding a valid solution in a short time. This does not mean that the optimal solution does not exist, but in light of this, we suggest starting future implementations by ordering the goal tasks.

Also, in a different test we made, we observed that using tasks as subtasks of methods increased the computational time. The first logical solution that we thought of was to unwrap the tasks by declaring in sequence each non-primitive task, but for a fashion look, we left them as tasks. In our code, we described step-by-step what to do if the reader would like to try it out. As an example, in the method M_deliver_supply_but_first_attach_carrier we call the task T_deliver_supply after executing the task to attach the carrier to the robot, which resulted, empirically tested, in a longer time required for the planner to find a plan w.r.t. the unwrapped version, but it increased the readability of the code.

4.4 Problem 4

For this problem, we analyze 3 different versions, and we want to find solutions that minimize the total time (in the code reported as (:metric minimize (total-time))).

4.4.1 With :negative-preconditions

The plan has been generated using the TFD planner, that supports the :negative-preconditions.

The solution is sub-optimal because the robot does not take all the advantage to fully load the carrier, and, at the

beginning of the plan, it delivers one box at a time. It is interesting to notice that, in a separate test we made, if we applied the condition not to load a box that has been already used for delivery, the final plan was reduced by one step. However, using the latter case broke the problem with only 3 boxes available, because we do not give the possibility to reuse the boxes.

We can conclude that the plan found in this case is not the optimal one. An extensive search would probably lead to better plans.

Extra problem with 3 boxes available We applied also in this problem the case where only 3 boxes are available. The plan has been generated using `TFD` again, and what catches our attention is the length of the steps. It is just one step more than the previous solution, underlining again that the main plan is sub-optimal. Also in this case we can expect a better plan with a deeper search.

4.4.2 Without `:negative-preconditions`

We created this version in order to use the `OPTIC` planner since it does not support `:negative-preconditions`.

What we immediately noticed with this planner is the amount of time that it takes to find the optimal plan. With this planner, a complete search to find the optimal plan is made. Unfortunately, after a couple of hours, the planner didn't provide any other better solutions, so we decided to stop the search and we relied on the last plan provided by the planner, the one reported in our repository.

The provided solution is shorter (three steps less) and faster (it takes one-third total-time less) than the one we got from `TFD` using `:negative-preconditions`. We think a better solution may exist, but a good trade-off between time and the goodness of the solution is important, depending on the application.

4.5 Problem 5

Using PlanSys2, we successfully generate a valid plan. The domain file is the same as the one tested in Problem 4 (Section 4.4.2), so we expected a similar plan as the one we got there.

As already mentioned in Section 3.5, the default planner is `POPF`, and we were not able to find options that allowed us to perform a complete search in the solution space. Thus, only one plan was returned, which was two steps longer than the one generated from `OPTIC` in Problem 4, so we can consider this solution as sub-optimal.

Observing the plan returned we can see that the robot fills the boxes and then delivers one box at a time, without considering loading multiple boxes and then delivering the content. We can argue that this solution is due to the short search space that has been made, returning the first plan that was valid. A deep search should provide better plans.

5. Conclusions

In this report, we designed, implemented and tested five different scenarios of an emergency services logistics problem.

We started from Section 1 to briefly illustrate the five tasks and how we set up our machines and then we described the general problem in Section 2, defining all the major assumptions at its base.

Then in Section 3 we described our solutions for every problem and how we managed the different issues encountered during the development. We have also reported lists of all the predicates, actions, tasks and methods for each implementation.

In the end, with Section 4, we analyzed the results obtained by running domains and problems with, if possible, different planners. Regarding Section 4, some implementations gave us optimal plans after a few iterations, while we got sub-optimal plans in the others. This was mainly due to some planners stopping the search for a goal when they found the first valid plan, without looking for a better one. In general, all the optimal plans we found follow the idea of preparing all the boxes first and then delivering them in one shot to the injured persons, if there are enough boxes available, without returning to the depot.

In Section 6 we discuss the archive's content and how it has been organized.

The HDDL implementation was the most critical one because we needed to take a bottom-up approach to build the code step-by-step in order to avoid bugs and possible issues that could prevent us from finding a valid plan.

Overall, we are satisfied with the results obtained and how we address problems encountered during the development of our solutions. This project can be used as a starting point for more complex applications in the area of automated planning techniques.

5.1 Future Works

For future implementations, it would be interesting to explore new alternatives and different approaches. These could include tweaking the durations in Problem 4, crafting varied hierarchical tasks for Problem 3, experimenting with a larger range of planners, especially in cases where only one plan was generated, and even testing the system on a physical robot using PlanSys2.

6. Archive overview & organization

In this section, we discuss the content of the archive, a downloaded version of our GitHub repository, and how it has been organized; the complete folder tree can also be seen in the GitHub README file.

First, the directory is organized into five different subfolders, each containing the code for one Problem. Inside the main directory, talking about files, we left the GitHub `README.md` and the pdf files containing this report and the assignment.

6.1 Sub-folder `problem1`

- `domain.pddl`: PDDL file for the domain of the basic problem
- `problem.pddl`: PDDL file for the problem

- `results.md`: markdown file used to save the machine setup and the results
- `runnerSolverAPI.py`: script that uses **planning.domains** API to solve the problem

6.2 Sub-folder `problem2`

- `domain.pddl`: PDDL file for the domain using the carrier to load boxes
- `problem.pddl`: PDDL file for the problem
- `problem_extra_3boxes.pddl`: PDDL file for the problem using only three boxes
- `results.md`: markdown file used to save the machine setup and the results
- folder `numeric_fluents`:
  - `domain.pddl`: PDDL file for the domain using `:numeric-fluents`
  - `problem.pddl`: PDDL file for the problem using `:numeric-fluents`
  - `runnerSolverAPI.py`: script that uses **planning.domains** API to solve the problem

6.3 Sub-folder `problem3`

- `domain.hddl`: HDDL file for the domain
- `problem.hddl`: HDDL file for the problem (with goal as ordered tasks)
- `results.md`: markdown file used to save the machine setup and the results
- `PANDA.jar`: PANDA planner Java file

6.4 Sub-folder `problem4`

- `domain.pddl`: PDDL file for the domain with durative actions
- `problem.pddl`: PDDL file for the problem
- `problem_extra_3boxes.pddl`: PDDL file for the problem using only three boxes
- `results.md`: markdown file used to save the machine setup and the results
- folder `no_negative_preconditions`:
  - `domain.pddl`: PDDL file for the domain modified to not use `:negative-preconditions`
  - `problem.pddl`: PDDL file for the custom problem

6.5 Sub-folder `problem5`

- `Dockerfile`: commands to assemble a Docker image for the PlanSys2 planner
- `setup.md`: markdown file that explains how to set the environment to run this problem
- folder `plansys_problem5`:
  - `CMakeLists.txt`: configuration file used by CMake to define build instructions
  - `launch_terminal1.sh`: bash file for the first terminal
  - `launch_terminal2.sh`: bash file for the second terminal
  - `package.xml`: manifest file used in ROS to provide metadata
  - `results.md`: markdown file used to save the results
- folder `launch`:
  - `problem`: problem file encoded as required for PlanSys2
  - `plansys2_problem5_launch.py`: script that selects the domain to run the corresponding executable that implements PDDL actions
- folder `pddl`:
  - `domain.pddl`: PDDL file for the domain with durative actions (the same as the one in the problem4/no_negative_preconditions directory)
- folder `src`: it contains all the `.cpp` files that are connected to the possible actions. They basically contain the source code that, in our case, prints out the action as well as the completion percentage. As already said before, in a real-world scenario, these files should contain the code to actually move a physical robotic agent and make it perform the required action.

References

[1] planning.domains API. [Online]. Available: https://api.planning.domains/

[2] Planutils repo and Docker Image. [Online]. Available: https://github.com/AI-Planning/planutils

[3] Docker Image for Java 8 'openjdk:8u342-jre'. [Online]. Available: https://hub.docker.com/_/openjdk/tags

[4] Java 8 recommendation. [Online]. Available: https://github.com/galvusdamor/panda3core

[5] PANDA Planner. [Online]. Available: https://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.090/panda/PANDA.jar

[6] ROS2 Planning System. [Online]. Available: https://plansys2.github.io/

[7] Stable Diffusion online for Image Generation. [Online]. Available: https://stablediffusionweb.com/#demo

[8] PDDL Extension for VS Code by Jan Dolejs. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=jan-dolejsi.pddl

[9] PlanSys Design Website Section. [Online]. Available: https://plansys2.github.io/design/index.html