Automated Planning: Planning Assignment

Kevin Depedri (mat. 229358)

University of Trento

kevin.depedri@studenti.unitn.it

1. Introduction

The purpose of this assignment is to implement a planning problem with gradual complexity over different planners. In particular, the assignment is organized in the following way:

- ***Task-1**, implementation of a basic version of the problem in* PDDL
- ***Task-2**, extension of the basic problem to a more complex implementation in* PDDL
- ***Task-3**, conversion of the extended problem to a hierarchical structure in* HDDL
- ***Task-4**, implementation of temporal domain and durative actions in the extended* PDDL *version of the problem*
- ***Task-5**, further extension of the last version of the problem, to simulate the performed actions through the run of action associated C++ codes*

To run different instances of the problem, more than one planner has been used. More in detail, the planners involved in the run of the different tasks are:

- ***Downward**, planner that supports* PDDL language*, used to run* Task-1 and Task-2
- ***PANDA**, planner that supports an hierarchical structure based on the* HDDL language*, used to run* Task-3
- ***Optic**, planner that supports time and durative actions, based on the* PDDL language*, used to run* Task-4
- ***PlanSys2**, planner that supports time, durative actions and that allows to simulate the execution of the plan. Based on the* PDDL language*, used to run* Task-5

This report is organized as follows. First, in Section.2, our interpretation of the problem for all the 5 tasks is defined. Then, in Section.3, the various design choices are described. Section.4 shows the result obtained running the tasks on the different planners, together with detailed comments over the obtained results. Finally, in Section.5, the conclusions over this work are followed by an outlook over possible future implementation that could be performed to extend this project.

All the documentation related to the project is available in the GitHub repository[1] present at the end of this page and in the final reference section. Furthermore, it is possible to find a comprehensive guide to install and run all the planners in the GitHub read-me file. This comes together with the results obtained by the planners over the different tasks.

2. Understanding of the problem

The assignment consists of modeling an emergency logistics service that is supposed to assist some injured people. The planning system will have to manage the activities of the robotic agents to deliver boxes containing emergency supplies to the injured people who need them. The main assumptions on which the problem is based are:

1. **Each injured person** is at a known a priori location (which is always different from the depot) and it does not move.
2. **Each injured person** has or does not have some specific supply. Therefore, each person might need nothing, just one specific supply, multiple supplies, or all of them.
3. **Multiple injured people** can be at a given location. Therefore, we will need to keep track of which supplies each person has/needs. Indeed, just keeping track of whether a box with supplies is arrived or not at a certain location is not enough.
4. **Supplies** are transported inside boxes and are modeled as box contents. Initially there are three types of supplies: *food, medicine and tools* which are all located at a specific location (the depot). Box contents are modeled in a generic way to allow new contents to be easily introduced in the problem.
5. **The boxes** are initially at a specific location (the depot) and they are empty and unloaded. They can be filled with one specific supply.
6. **Robotic agents** are the workers appointed to assist the injured people. We have a specific type called *robotic agent* which will have just one instance in all the previously defined tasks. This robotic agent is initially at a specific location (the depot).
7. **Filling a box** can be performed by robotic agents if the box is empty and if the agent, the box and the supply are all at a same location. The result is a box filled with that supply.
8. **Loading or unloading a box on the robot** can be performed by robotic agents if the agent and the box are in the same location. The robotic agents have a *carry capacity of 1 box* for all the defined tasks. Therefore, it can be loaded only if it is not loaded yet, and it can be unloaded only if it is loaded.
9. **Robotic agents can move directly** between arbitrary locations. The locations are all connected between them as in a fully connected graph. Therefore, there is no road map that need to be followed to reach some specific locations. When a robotic agent move, if it is loaded, also the loaded box moves with it.
10. **Unfilling a box** can be performed by robotic agents if the box is filled with a supply and if the agent, the filled box and the injured person who needs this supply are all at a same location. In that case the box is unfilled and the content is given to that person.

[1] GitHub Repository link

In addition, later tasks define the following additional assumptions:

1. **Carriers** are vehicles which can be driven by a robotic agent to deliver more boxes in an efficient way. We have a specific type called *carrier* which will have just one instance in all the tasks starting from *Task-2*. There is no carrier in *Task-1*. This carrier is initially at a specific location (the depot).

2. **Carrier capacity** is problem specific. Therefore, it is defined in the problem file using a predicate for each loading position (i.e. position to load a box) that we want to have on a particular instance of carrier. It is not possible to load on the carrier more boxes than the one specified by its capacity. Furthermore, loading positions allow to keep track of how many boxes are currently loaded and how many boxes can still be loaded.

3. **Loading or unloading a box on the carrier** can be performed by robotic agents if the agent, the box and the carrier are in the same location. The robotic agent will load the carrier in the first available loading position. Therefore, it can be loaded only if there are free loading position, and it can be unloaded only if there is at least one loading position which is not free. The number of loading positions is defined in the problem file as seen from the previous point. Furthermore, a major assumption is that the robot who drives the carrier need to be loaded, in this way the carrier is used only in cases where it is really needed. This allows us to transport an additional packaged w.r.t the carrier capacity (e.g, if carrier capacity is 4, then, considering the box loaded on the robot who drives it, we can move up to 5 boxes).

4. **Carriers can move directly** between arbitrary locations when a robotic agent drives them. When a carrier moves, also the robotic agent driving it and the loaded boxes move with it (both boxes loaded on the carrier and the box loaded on the robot). A major assumption is that carriers are never moved empty, for this reason, a carrier can be moved only if the robot who drives it is loaded, and only if also the carrier is loaded with at least one box.

5. **Action associated C++ codes** are based on the concept of 'fake action'. It means that the C++ codes just return some incremental information about the procedure in progress during its completion time. It is possible in any moment, if wanted, to modify these codes, improving the actions performing something more elaborated.

3. Design Choices

In this section the design choices performed for each one of the previously defined tasks are discussed.

Furthermore, for each task, a list of all the involved actions and predicates is given. The names of the actions and of the predicates present in these lists are quite self-explanatory, for this reason, to keep this report as concise as possible, they are not commented here. Anyway, a deeper description and explanation of every action and predicate listed can be found looking at the *PDDL or HDDL domain-file* of the respective task.

3.1. General design choices

All the following tasks are based on the following general design choices:

- **Injured people**, we suppose to have 2 injured people *p1 and p2* which are respectively in location *l2 and l4*. Injured person *p1* has nothing and *needs both food and medicine, but not tools*, while Injured person *p2* has both food and medicine and just needs *tools*.
- **Locations**, we suppose to have 5 locations, *l0, l1, l2, l3 and l4*. Of these, *l0 is the depot*, this means that initially, robotic agent, supplies, carrier (see *Task-2*) and boxes will be located here. Furthermore, as previously stated, no *injured person* can be at the depot.
- **Supplies**, the quantity of supply present at the *depot* is supposed to be unlimited. Therefore, the action of filling the boxes will not deplete the stock of resources available at the depot. The available supplies are *food, medicine and tools*.
- **Boxes**, we suppose that each box can be filled with just one supply. Therefore, it can be used to satisfy the needs of *just one injured person*. Furthermore, we assume that boxes need to be reused. For this reason, each box need to be returned to the *depot* after being unfilled. In addition, we want all the boxes at the depot to be be empty and unloaded when not used. In this way the boxes will be ready for the next emergency call in the smallest possible time.
- **Robotic agent**, the robotic agent *r1* is initially empty and it is located at the *depot*. It has a *carry capacity of 1 box* for all the defined tasks. In addition, we want the robotic agent to be at the depot and to be unloaded when not used. In this way the robotic agent will be ready for the next emergency call in the smallest possible time.
- **Carrier**, the carrier *c1* is initially located at the *depot*. The carrier has a carry capacity of 4 boxes and it is initially unloaded in all these loading positions. In addition, we want the carrier to be at the depot and to be totally unloaded when not used. In this way the carrier will be ready for the next emergency call in the smallest possible time. The carrier is introduced starting from *Task-2*.

3.2. Task-1

In *Task-1* we use the base setup of the problem which has been previously defined, with the following additional design choices:

- **Robotic Agents**, as stated previously this task leads to the introduction of the robotic agent *r1*.
- **Boxes**, in this first instance of the problem we assume to have just one box *b1*. This is more than enough, since, we can carry just one box at time (we have only one robotic agent) and since the box needs to be reused.

The list of actions implemented in *Task-1* is:

- Move unloaded robot
- Move loaded robot
- Load robot
- Unload robot
- Fill box
- Unfill box

The list of predicates implemented in *Task-1* is:

- Location
  - Injured person located at
  - Robotic agent located at
  - Box located at
  - Supply located at
- State of injured people
  - Injured person has supply
  - Injured person has not supply
- State of boxes
  - Box has supply
  - Box has not supply
  - Box is unloaded
  - Box is loaded
- State of robotic agents
  - Robot is unloaded
  - Robot is loaded

3.3. Task-2

In *Task-2* we extend the problem of *Task-1* with the following additional design choices:

- **Carriers**, as stated previously this task leads to the introduction of the carrier *c1*.
- **Boxes**, in this second instance of the problem we assume to have five boxes *b1, b2, b3, b4 and b5*. This is more than enough, since, now we can carry up to 5 boxes at time (4 on the carriers and one on the robotic agent who drives it) and since the boxes needs to be reused. Notice that this number of boxes is kept the same for all the next tasks since they are all based on the same configuration of robotic agents and carriers.

The list of action implemented in *Task-2* is composed of the actions implemented in *Task-1* (see above) with the addition of the following new actions:

- Move 1 loaded carrier
- Move 1-2 loaded carrier
- Move 1-2-3 loaded carrier
- Move 1-2-3-4 loaded carrier
- Load carrier position 1
- Load carrier position 2
- Load carrier position 3
- Load carrier position 4
- Unload carrier position 1
- Unload carrier position 2
- Unload carrier position 3
- Unload carrier position 4

The list of predicates implemented in *Task-2* is composed of the predicates implemented in *Task-1* (see above) with the addition of the following new predicates:

- Location
  - Carrier is located at
- State of boxes
  - Box is carrier loaded
- Carrier capacity
  - Carrier capacity 1 boxes
  - Carrier capacity 2 boxes
  - Carrier capacity 3 boxes
  - Carrier capacity 4 boxes
- State of carrier
  - Carrier position 1 free
  - Carrier position 2 free
  - Carrier position 3 free
  - Carrier position 4 free
  - Carrier position 1 loaded
  - Carrier position 2 loaded
  - Carrier position 3 loaded
  - Carrier position 4 loaded

The actions and the predicates related to the carrier (loading, unloading and moving it) have been defined up to 4 loading positions, as previously stated in the assumption. This means that with these actions/predicates it is possible to define fully working carriers with a carry capacity up to 4 boxes. If we are interested in defining bigger carriers, it will be necessary to add new actions and predicates to deal with the newly introduced loading position. This can easily been accomplished just taking the structure defined up to now and extending it.

3.4. Task-3

In *Task-3* we convert the problem of *Task-2* to the Hierarchical Domain Definition Language (*HDDL*). This allows us to address the problem using an Hierarchical Task Network (*HTN*). To do so, we introduce the following goal tasks which are then used in the goal section to define what we want the planner to accomplish:

- **Goal tasks**
  - Deliver 1 supply to injured person
  - Deliver 2 supply to injured person
  - Deliver 3 supply to injured person

The previously defined goal tasks rely on the following list of tasks, which then rely on the ensuing list of methods and on the actions implemented in *Task-1 and Task-2* (see above):

- **Tasks**
  - Get robot to location
  - Get carrier to location
  - Robot fill box
  - Robot unfill box
  - Robot load robot
  - Robot unload robot
  - Robot load carrier
  - Robot unload carrier

- **Methods to deliver supplies**
  - Deliver supply by robot
  - Deliver supply by adding to carrier
  - Deliver supply by carrier 1
  - Deliver supply by carrier 2
- **Methods to get robot to location**
  - Robot is already there
  - Robot unloaded moves to location
  - Robot loaded moves to location
- **Methods to get carrier to location**
  - Carrier is already there
  - Carrier 1 loaded moves to location
  - Carrier 2 loaded moves to location
  - Carrier 3 loaded moves to location
  - Carrier 4 loaded moves to location
- **Methods to fill boxes**
  - Box already filled
  - Robot fill box with supply
- **Methods to unfill boxes**
  - Box already unfilled
  - Robot unfill box with supply
- **Methods to load robot**
  - Robot already loaded
  - Load robot
- **Methods to unload robot**
  - Robot already unloaded
  - Unload robot
- **Methods to load carrier**
  - Carrier already loaded
  - Load carrier position 1
  - Load carrier position 2
  - Load carrier position 3
  - Load carrier position 4
- **Methods to unload carrier**
  - Carrier already unloaded
  - Unload carrier position 1
  - Unload carrier position 2
  - Unload carrier position 3
  - Unload carrier position 4

Notice that all the methods which have in their name the world *"already"* make use of *no-op actions*. These are empty actions, which allow to call the method that implements them when a condition that we want is already satisfied. In this way, we are allowed to move on with the linear progress of a task without performing any action since this condition is already achieved from the beginning. To see all the list of no-op actions please refer to the *Task-3 HDDL domain-file*.

3.5. Task-4

In *Task-4* we implement the temporal domain, converting all the actions defined in the problem instance of *Task-2* to durative actions. To do so, all the actions have been improved defining a time duration, and some time constraints which need to be satisfied *at start, over all and at end* of the actions. These constraints allow to avoid overlaps in time for actions that, according to our implementation, cannot be executed in parallel. The result is the following set of actions with the specified duration:

- **Move unloaded robot**, duration 1
- **Move loaded robot**, duration 3
- **Load robot**, duration 1
- **Unload robot**, duration 1
- **Fill box with supply**, duration 2
- **Unfill box with supply**, duration 2
- **Move X loaded carrier**, duration 3
- **Load carrier position X**, duration 2
- **Unload carrier position X**, duration 2

In the list above the letter *"X"* has been used to point that, all the actions with that same name that just differ by a number (in the place of X) share the same time duration.

One thing to notice is that, due to how the problem has been designed, none of the actions defined in the list above can be executed in parallel by a robotic agent. Therefore, time constraints have been set up to avoid this behavior and a new predicate has been introduced. Furthermore, since the planner used to deal with durative actions (*Optic*) does not support negative preconditions, 4 new predicates have been introduces to describe the state of the carrier's loading position. Therefore, the list of predicates implemented in *Task-4* is composed of the predicates implemented in *Task-2* (see above) with the addition of the following new predicates:

- **Additional predicates for temporal planning**
  - Robot is not busy
  - Carrier position 1 not free
  - Carrier position 2 not free
  - Carrier position 3 not free
  - Carrier position 4 not free

The first predicate is normally false and it becomes true as soon as an action that requires the robotic agent starts. This prevents the robot from performing more than one action at time. The other 4 predicates are just the negation of the predicate *Carrier position X loaded*. In this way we are able to verify with a positive precondition that one place is not free, avoiding to load on that position and loading on the closest position available.

3.6. Task-5

In *Task-5* we further extend the last version of the problem from *Task-4* to simulate the duration and the execution of actions through the run of action associated C++ codes. In particular, a C++ code has been created for every action defined in the previous list of actions (see *Task-4*). These codes have the same duration as defined above and are implemented as a *"fake action"*. This means that they will give an update on the state of the process each 250ms, without performing anything more. Also in this case we need to avoid parallelization and we do not have support for negative preconditions (in PlanSys2). Therefore, the additional predicates defined above are retained.

4. Results

In this section the resulting plans obtained for each one of the previously defined tasks are discussed. These plans are present into the *computed plans* folder on the GitHub repository. Furthermore, they are also shown and quickly commented at the end of the *GitHub read-me file*.

4.1. Task-1

The plan related to *Task-1* has been computed using *Downward* planner from the *Planutils* suite. To perform a complete search in the solution space we decided to use the *alias lama* instead of the *alias lama-first*. In this way we have been able to obtain gradually better versions of the plan when they existed. In the case of *Task-1*, we can affirm that the first obtained plan is an optimal plan simply looking at its structure. Here we have the robot that keeps on: filling a box, loading it, delivering it, unfilling it, loading it again and bringing it back to the depot. This sequence is repeated for all the supplies that need to be delivered. This solution is confirmed to be optimal also by the planner, since no better solutions can be found, leading to terminate the search procedure in a short time.

4.2. Task-2

The plan related to *Task-2* has been also computed using *Downward* planner from the *Planutils* suite. Also in this case we used *alias lama* to obtain gradually better versions of the plan when they existed. The first plan computed for *Task-2* shows a sub-optimal behavior, this is quickly addressed by second plan which is almost optimal. Finally, about 15 minutes later a third version of the plan has been computed. We can affirm that this last version of the plan is an optimal solution, achieving the goal state in the smallest possible number of actions. Here we have a correct use of the carrier, which is loaded in the best way possible, and which is moved only when necessary, avoiding the useless and inefficient actions which were present in the first two versions of the plan. The search space resulting by this version of the problem results to be quit massive. For this reason the solution is not confirmed to be optimal by the planner, because, after 2 hours of computation a crash due to insufficient memory has occurred, leading to terminate the search procedure. Notwithstanding that, looking at the plan it is easy to understand that a shorter plan cannot be achieved, making this plan an optimal solution.

4.3. Task-3

The plan related to *Task-3* has been computed using *Panda* planner. Unfortunately, on this planner there are no options like in *Downward* that allow to perform a complete search in the solution space. For this reason only a plan can be returned and then the search is terminated. Initially we obtained a plan that is sub-optimal, since it requires a higher number of actions with respect to the one computed in *Task-2*, also if the problem and the goal are still the same. For this reason we tried to comment the method *deliver by robot*, which, in our opinion, was at the base of this inefficiency. After that we performed a second search and we achieved an optimal plan, with the same length as the one computed in *Task-2*. This confirms that indeed the optimal solution existed, but that simply it was not found since the planner stopped its search at the very first viable plan.

While analyzing the results of this task we need to remember that *Task-3* is based on a structure which is very different from the one of *Task-2*. Indeed, here the problem and the domain files have been written in *HDDL*, for this reason we had to do numerous design choices to allow the *Panda* planner to achieve the same result as the one achieved by the *Downward* planner. In particular, the main issues related to the conversion *from PDDL to HDDL* have been:

- **HDDL goal**, it is coded differently with respect to PDDL. Indeed, in PDDL we describe the state that we want to achieve, and the planner will find the best actions to achieve it. Instead, in HDDL we need to define which tasks (goal tasks) we want to perform and the planner will choose the best methods to achieve them. For this reason we had to code multiple goal tasks, for the different numbers of supplies that an injured person could require.
- **HDDL tasks**, these tasks are based on multiple components, which generally are other tasks and methods. All the components present in a task (or in a goal task) need to be performed. But, according to the case, this is not always the behaviour that we would like to experience. Indeed, some components of the tasks could result redundant or useless in certain scenario. For this reason, to optimize the execution, we are forced to define many different methods which differs between them just for a few lines. This makes the code very long and difficult to understand.
- **HDDL ordering and merging**, generally, each task in HDDL defines an ordering which need to be followed in executing its components, this lowers the task's flexibility. Furthermore, also in the goal we are supposed to define an ordering. In this case, if the ordering is imposed, each goal task will need to be achieved before moving on the next one. If that is the case, it would be impossible to integrate different tasks together to reach an efficient solution (e.g., different supply need to be delivered according to different goal tasks, if the ordering is imposed then it would be impossible to load the carrier efficiently, since we would need to fully deliver a box before moving to prepare the next one). For this reason, we decided to not implement the ordering in the definition of the goal. This allowed the Panda planner to aggregate together the methods used inside the different goal tasks, reaching a solution which is way more efficient than the one that can be achieved imposing a goal ordering.
- **HDDL task initial check**, a good practice in HDDL tasks is to verify, with one or more initial methods, that every instance is in its correct state before starting with the execution. For example, we should verify that the robot is at the depot, that the boxes are at the depot, and so on (and if they are not there, we should take actions to bring them there). In our case the initial scenario is already correctly aligned to allow any possible action. Furthermore, each task is designed to put every instance in the same state where it was, before reaching its end. For this reason we decided to not implement these check, otherwise our solutions would be filled with no-ops.

All these previous points make it very difficult to achieve the same behavior between a PDDL implementation and an HDDL one. For this reason many repetition and many assumptions are needed. Please check the *Task-3 codes* for further insights.

4.4. Task-4

The plan related to *Task-4* has been computed using *Optic* planner from the *Planutils* suite. The metric used in the problem file to look for a solution is *minimize total time*. The first plan that we obtained is sub-optimal, since it requires a longer time (2 seconds more) than the lowest possible theoretical time for our scenario. This is due to two useless actions performed in row (unloading and loading a robot without doing nothing in between), which are not necessary since they do not bring any change in the environment. At this point we decided to let the planner run to see if the optimal solution could be found. Indeed, in this case we cannot rely on commenting the part of the code that caused the plan to be sub-optimal, as we have done in *Task-3*, since, this part of the plan (*load robot* and *unload robot*) is absolutely necessary to perform the other actions and to get a plan. The results of letting the planner run is that as *Downward in Task-2* it ran out of memory after a few hours of search. This does not mean that the optimal solution does not exist, it simply means that we have not enough memory to find it. This was quite expected since computing a better solution is a much more heavier task than computing the first one.

4.5. Task-5

The plan related to *Task-5* has been computed using *PlanSys2* planner based on the *ROS2* infrastructure. Unfortunately, in this planner as in the Panda planner there are no options that allows to perform a complete search in the solution space. For this reason only a plan can be returned and then the search is terminated. *Task-5* is based on the exact same *domain PDDL file* of *Task-4*, for this reason we expect a similar solution to be computed. Again, the plan that we obtained is sub-optimal, requiring the same exact time as the plan computed in *Task-4*. This is due to the same two useless actions (unloading and loading a robot), which in this case have not been performed in row, but that do not contribute to the solution in any way. Also in this case we cannot comment these two actions, but we can with confidence associate this behavior to a search in the solution space that has been too short.

5. Conclusion

In this report we have discussed the implementation of a problem divided in tasks with gradual increasing complexity. In Section.2 we described the problem over all its dimensions, defining clearly all the major assumptions at its base. In Section.3 all the performed design choices have been listed and explained, together with a comprehensive list of the actions and predicates used in each task. Finally, in Section.4, the results obtained over the different tasks have been commented, highlighting which were the possible causes at the base of sub-optimal plans and taking action to fix these unwanted behavior when it was possible. Here we have seen how, results ranging from optimal to slightly sub-optimal depended mainly on the planner search strategy, and, when possible, we commented parts of code that were not strictly necessary, to drive the search in our wanted direction, proving that an optimal solution existed. In this section we have also discussed the main implementation issues that have been encountered while writing the codes for this project. Between all the possible issues, the HDDL implementation clearly stands out as the most complex task, which requires an huge quantity of assumptions and of design choices to have a behavior as the one of its respective PDDL implementation.

Section.4 ends with the discussion of the plan returned by PlanSys2, and it is here that, according to us, eventual future works could be focused on. Indeed, in Task-5 we just asked PlanSys2 to compute a plan and then we ran ”fake actions”, but it would be great to move from a fake-action implementation to an implementation with real actions. In that case, the plan computed would be effectively used to run codes in a meaningful order, and further considerations, not only on the planning side, but also on the domain implementation side should be done to ensure an efficient and error free implementation.

6. References

[1] [Rov22] Marco Roveri. Assignment for the course Automated Planning Theory and Practice. Academic Year 2022-2023, Dec 2022.

[2] [Project GitHub] Kevin Depedri. Automated Planning - Planning Assignment repository. Academic Year 2022-2023. [https://github.com/KevinDepedri/Automated-Planning] GitLink