Automated Planning

Bussola Riccardo ID: 240080, Zilio Nicola ID: 240187

23 December 2023

0.1 Analysis of the problem

This section briefly discusses the analysis produced to understand the requirements and develop a solution for the proposed problem. Since different parts of the assignment are purposely under-specified, we will explain the assumptions and implementation choices adopted.

Approach followed

For each of the proposed problems, we have methodically proceeded as follows: extract the key elements of the problem and their corresponding functionalities, extract the relevant actions required to accomplish the specified goal, build the action according to our assumptions and implementation choices, add additional predicates to manage different requests for each action. To avoid compatibility issues between the adopted planners, we've implemented our solutions following the "as simple as possible" philosophy. Despite our awareness of the possibility of introducing some redundancy, our solutions guarantee effectiveness.
The foundation for our action descriptions lies in a set of fundamental predicates, representing essential concepts and relationships within the domain. These predicates are built to be generic and can be extended with problem-specific ones, such as we will see: robot capacity or maximum payload constraints etc. Hereafter, we will present the predicates we have determined to be suitable and explain the factors that influenced our decision.

- **(atws ?x - work_station ?l - location)**: Defines in which *location* a *work_station* is located, since the problem says that there can be more than one *work_station* per *location* this allows us to ensure this property.
- **(atbl ?x - box ?l - location)**: Defines where a *box* is, since we have to keep track of where boxes are located, we may change all the locations of the boxes with this predicate
- **(atbw ?x - box ?l - work_station)**: Defines if the *box* x is associated to a specific *work_station*, since being at a location l does not mean to be assigned to a specific *work_station*, then if we want to say that a specific *box* is assigned to a *work_statio*n we will use this predicate
- **(ats ?x - supply ?l - location)**: Defines where a specific *supply* is located, this is useful at the beginning when we want to define that all the supplies are located at the *centralwarehouse*.
- **(atr ?x - robot ?l - location)**: Defines where a *robot* is located, it is used to keep track of the position inside of the map of the *robot*.
- **(contains ?x - box ?y - supply)**: Defines the content of a specific *box*, this is used to track the content of a specific *box*, it is made as general as possible to be able to have inside any type of *supply*
- **(has ?ws - work_station ?s - supply)**: This predicate is used to specify that a certain *work_station* has been provided for a certain supplysupply*supply*, not knowing the type of the *supply* that has been delivered, this paired with the class-checking predicates that will be explained later, will allow us to achieve the goal
- **(is_bolt ?b - supply), (is_valve ?b - supply), (is_tool ?b - supply)**: These predicates are the class-checking we introduced before. These are used to check if a general *supply* is an instance of a specific *supply* class *(bolt, tool, valve)*.
- **(has_bolt ?ws - work_station), (has_tool ?ws - work_station),**
  (has_valve ?ws - work_station): These are the predicates that we will use as the "goal" definer, this predicate is the one that has the objective of stating that a specific *work_station* needs certain *supply*, hence the goal will be to provide the *supply*.

- **(box_delivered ?b - box ?ws - work_station)**: Defines if a *box* has been delivered to a specific *work_station*, this is needed since some actions may be done only if the *box* has been delivered.
- **(carries ?r - robot ?b - box)**: Defines if a *robot* is carrying the *box* b, this is used to assign a specific *box* to a *robot*.
- **(connected ?l1 - location ?l2 - location)**: This specific predicate is used to define that there is a connection (hence a road) between two locations. This predicate is not intended to be bi-directional, hence if (connected l1 l2) is true, then (connected l2 l1) may not be true.
- **(free ?l - location)**: Defines if a specific *location* is free, this predicate is used to avoid having two robotic agents at the same time on the same *location*.
- **(loaded ?r - robot)**: Defines if a *robot* is loaded or not. This predicate is used to determine if we have to move only the *robot* or the *robot* and the boxes.
- **(empty ?b - box)**: Determines if a *box* is empty or full. If a *box* is full then it will be impossible to fill it again, this means that we are ensuring that every *box* may carry only one supply

Throughout the whole development of the project, whenever we were dealing with something that was not clear or left unspecified, we decided to try to have as many interactions as possible with the client who was requesting the system (the professor) to clarify any possible doubt and in case to find a compromise in the solution.

Problem 1

Starting from the description of the problem we have identified the following key elements of our environment: *locations, workstations, robots, and supplies*. Those elements are then further explored and enriched to account for the newer system's needs. To manage the supplies abstractly, we have declared a general-class *supply* on top of which we can define more specific types such in our case: *bolts, valves, and tools*. For what regards locations, the only particular decision we have made in terms of implementation was to dedicate a special type for the warehouse, this was done to manage easily some behavior concerning this position.

Hereafter, we will present the implemented actions and their roles in the problem:

- **move_robot**: It is the standard action to move a *robot* from one *location* to another. This action is meant to be used only when we want to move an unloaded robot, hence no boxes are moving with it.
- **move_box_and_robot**: This is a modification of the standard move action that allows moving both a *robot* and a *box* at the same time keeping track of their change of positions. This action hence requires the *robot* to be loaded with one *box*.
- **pick_box_from_location**: This action is meant to be used to collect one *box* from a certain *location*, this assumes the *robot* to be in the same *location* of the *box* and to be able to load it. As an effect the *box* will be associated with the *robot* and the status of the load of the *robot* will be modified.
- **deliver_box_to_ws**: The role of this action is, after a *robot* arrives at a certain *location* with some load, it can discharge the load to a specific *workstation*, give that the *workstation* is in the same location as the *robot*. This will produce the fact that the *workstation* will now have a *box* with certain content.

- **fill_box**: This action is meant to be used to fill a *box* with a generic *supply*, this has been made general in order not to have to create a single action for each one of the available supplies.
- **empty_box**: This action is meant to represent a *robot* to empty the content of a *box* and to associate the content with the *workstation*. This has in fact as a precondition that the *robot* is at the same *location* of the *workstation* and that a *box* containing some supplies has been delivered.
- **bolt_acquired , tool_acquired , valve_acquired**: These actions are meant to be used as actions' confirmation. The way we thought to use them as flags was to determine when a certain *workstation* acquired a specific *supply*. The effect of these actions is to set *has_supply* (with *supply* changes with bolt, valve, or tool) at **true**. In our mind, those predicates that will be set as **true** by the effect of those actions are the ones that will be used to define the goal.

Problem 2

The second problem required to extend the first problem allowing the possibility to include various types of robotic agents, each of them characterized by a capability, that is specific for each type of agent and that specifies the maximum number of boxes that a specific agent can carry. Since some of the planners do not allow numerical representation, as suggested in the lectures we have implemented the counting method by exploiting Peano's axioms. Hence we defined a new type *number*, that will be a placeholder for a number, and we then defined the predicated *successor and predecessor* that will build the relationships between the various placeholders to build the counting system. Once defined the counting system we have to define a way to assign a capacity to each robotic agent. To accomplish this need we proceeded as follows:

- **Defining minimum capacity**: For each of the agents we introduced a predicate called *min_capacity*. This predicate will define the lower bound of each robotic agent's capacity. When modeling the problem this will be set up as the placeholder for the number 0.
- **Defining maximum capacity**: For each robotic agent, we decided to create a predicate called *max_capacity*, this will allow us to set the desired maximum capacity of each agent. A possible implementation of this predicate would have been the association of a maximum capacity to each class of agents. Acting this way, however, would have resulted in a loss of degree of freedom in terms of problem-specific generalization. Our implementation choice allows adaptability to different scenarios where the same class of agents have different capacities.
- **Modify previous actions to include the counting mechanism**: The only actions that should be modified to include this mechanism are the moving actions and the load/unload actions. The moving actions have been modified to take a capacity value as the input parameter. Depending on the action, the *min_capacity* predicate is evaluated: if it is true, then the robot is unloaded, hence we move only the robot, otherwise, we move the robot with the boxes.
  Concerning the loading/unloading of the boxes, the two of them have similar but opposite behaviors: they both take as input two capacities, which will be the capacity before and after the action. If the bounds on the capacities are broken by doing the action (the capacity is equal to min_capacity for the unload and max_capacity for the load) then the action cannot be executed. If the action can be executed, then the actual capacity of the robot will be updated.
  To allow this to happen, we had to introduce a new predicate that keeps track of the actual capacity of each robotic agent, this is done by the *(*capacity) predicate, which takes as input a robot and a capacity.

Problem 3

The third problem is stated from the same scenario that serves as the basis for problem 2, this time however, we have to address the problem of exploiting Hierarchical Task Networks (HTN).
The first task was to create a domain similar to the one we had in problem 2. Once this domain was created, we took a closer look at the actions to see if two or more actions could be joined together to start building the new hierarchical tasks. This was made starting from the lower level until the higher level task was reached. In our specific case, this consists of delivering a specific supply to a workstation. The tasks that we have modeled have the following structure:

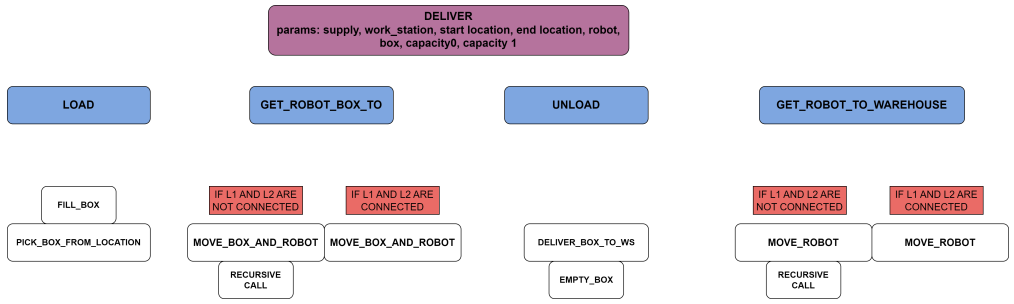

Figure 1: Task hierarchy

As you can see from the picture, the tasks we have defined on the first level of the hierarchy put together actions to perform general stuff such as loading and unloading a robot, with all the single actions that may be required to act. For the part that has the goal to move a robot from one location to another, the tasks are a little bit more complicated. To move a robot from one location to another we may have two cases: the first one is when the two locations we are considering are connected, this means we don't have any intermediate step between the two locations and we can move without any particular constraint. On the other side, if the two locations are not directly connected but there are some other locations between them, what we have done is use some preconditions on the action, hence checking if a possible via-point connecting the two locations corresponds to the final point. If it does not (hence the locations are not connected) we create a recursive task that will deal with moving the robot between the necessary via-points until the final destination is reached. In addition to this, since in our modeling of the simple actions we defined two possible cases in which the robot can be moved (they depend on the fact that a robot is or isn't carrying some boxes), we were forced to create two different tasks based on the final action we needed to perform. Another problem that was occurring was that, as the problem is requiring, the warehouse must not contain any work_station needing something. Since we don't have a reference for the work_station in the warehouse, we created a specific task that is used to take back the robotic agent to the central warehouse.

Problem 4

The fourth problem required modifying the domain of problem 2 to allow the usage of durative actions. To achieve this condition, we started from the previously defined actions, and by splitting the preconditions we built the ones needed for this problem. Since one of the requirements of the problem was the possibility of performing actions in parallel for different agents (so, an agent can perform a single action at a time), we have introduced a new predicate that allows us to define whether a robotic agent is busy or not. Before starting an action, we exploit this predicate to check if it is possible to execute an operation. If the agent is free, at the start of the action we set the *busy* predicate to true, this way the robot will be considered occupied for the whole duration of the action. When the action reaches the end, the predicate will be negated, hence, the robot will be free to start new actions.

Problem 5

The fifth problem required us to bring everything we implemented with durative actions, into the ROS2 planning system. The first thing to do was to configure the *plansys2* environment to work, then once the workspace was built, we proceeded to start from the examples we have seen in the lectures. The *simple_example.pddl* file has been modified to be compliant with what we have done in the previous problem. What we did was copy everything in the file and try to run the simple planner. Since the POPF planner was creating problems we decided to use the Temporal Fast Downward planning package for ROS2 plansys [1]. Once we knew that the planner was able to find a solution for the planning problem we were providing, then we started building the fake actions. Since the duration of the actions has to be according to the one that we assigned in the pddl domain, what we did was to modify the duration values inside of the for loop that is creating the progress of the fake action. Since the check on the action is done every 100ms, then the for loop that is tracking the progress of the action will be updated in a way such each step of the progress will be defined by 100ms divided by the total duration of the action, hence the progress of the action will have a duration that corresponds to the total effective duration of the action when applied to the final plan. To make things faster to run on the plansys2 terminal, we created a file containing all the commands we should have written inside the terminal to a file called *commands* that we can source after in the terminal to load up the problem, then we can define the goal of the problem we want to solve inside the terminal and then execute the plan.

0.2 Evaluation of each problem

For each of the five problems we defined above, we decided to test how the planner behaves with three different problems of increasing complexity. Hereafter we will shortly describe the problems we decided to use and we will make considerations on the duration time of the plan and the results for each task.

Problem 1

The first problem was evaluated using different planners to solve three problems with increasing complexity.
The first configuration is a simple map with 4 locations, 1 central_warehouse, and 5 workstations. We suppose that at the beginning, all the supplies and the boxes are at the warehouse. In this particular problem, we decided to use 6 boxes and 6 pieces of supply for each class.
The goal for the first problem was to deliver four supplies to various *work_stations* in the map.

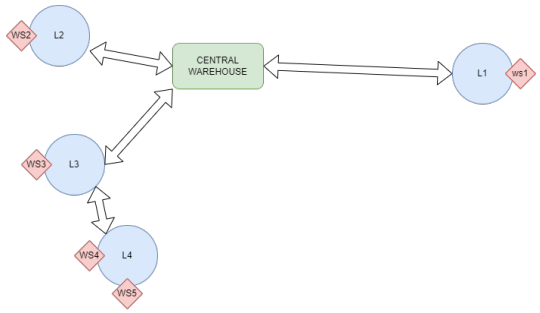

Figure 2: Map of level 1

Planner	Total Time(s)	Evaluated States	Path Length
Downward	0.051765s	218	34
ff	10.36s	74796	28
symk	0.0511894s	218	34
lpg-td	0.51s	not provided	28
lama	0.0148244s	218	34

Table 1: Results of problem 1 configuration 1 with different planners

Then we proceeded to evaluate with different planners the second level of complexity for our problem. The configuration of this problem consists of 6 locations connected as shown in Fig. 3 with 10 workstations.
The goal for this specific problem is to deliver 9 supplies to various *work_stations*.

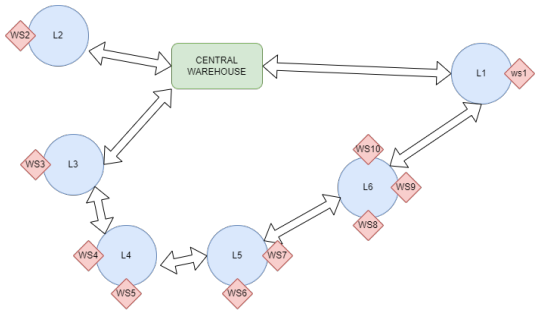

Figure 3: Map of level 2

Planner	Total Time(s)	Evaluated States	Path Length
Downward	1.02507s	3321	82
ff	0.60s	3422	76
symk	0.857072s	3321	82
lpg-td	16.65s	not provided	116
lama	0.830554s	3321	82

Table 2: Results of problem 1 configuration 2 with different planners

In the end, the last level of complexity we tested our problem on was a configuration in which we had 14 locations and 16 work_stations, and the robot had to perform 15 deliveries.

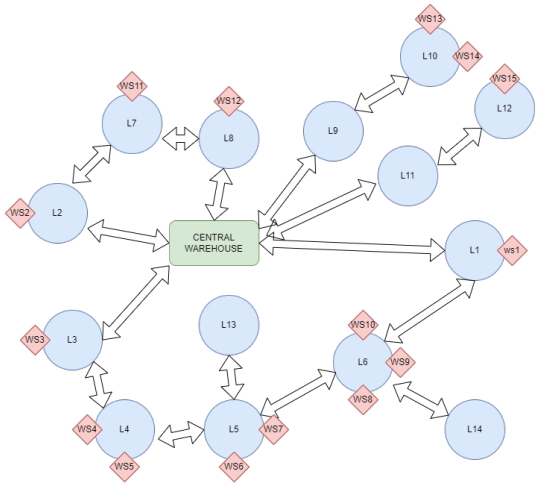

Figure 4: Map of level "extreme"

Planner	Total time(s)	Evaluated States	Path Length
Downward	12.1363s	3532	188
symk	10.9714s	3546	188
lama	12.2507s	3532	188

Table 3: Results of problem 1 configuration 3 with different planners

The results of the evaluation of the first problem are quite straightforward, in fact in the majority of the cases *Downward*, *Lama* and *symk* are outperforming the other planners. There may be some exceptions like in the second configuration, where the *ff* planner can outperform the others both in terms of the time to find a solution and in the path length. On the other side, it may happen that for simpler configurations such as the first one, *ff* takes way longer than the others and the number of evaluated states is way bigger than the others despite having the same path length of the *lpg-td* planner. In the most complex context, we can highlight that both *ff* and *lpg-td* are not able to find a solution (hence we did not even report them), on the other side, we can see that all the other three planners can find a solution, with *symk* slightly performing better than the other two in terms of total time having the same path length.

Problem 2

To evaluate problem 2 we had proceeded in the same way as we proceeded for problem 1. We evaluated the domain we created with different problems of increasing complexity (for this problem, they are the same as problem 1 but with a changed type/number of robotic agents)

Planner	Total time(s)	Evaluated States	Path Length
Downward	0.0086968s	68	25
ff	0.01s	218	35
symk	0.0089772s	68	25
lpg-td	4.02s	not provided	37
lama	0.0088304s	68	25

Table 4: Results of problem 2 configuration 1 with different planners

Planner	Total time(s)	Evaluated States	Path Length
Downward	0.336757s	2386	90
ff	9.47s	30914	87
symk	0.563435s	1427	91
lpg-td	30.28s	not provided	128
lama	0.913971s	2326	73

Table 5: Results of problem 2 configuration 2 with different planners

Planner	Total time(s)	Evaluated States	Path Length
Downward	12.9582s	3300	140
symk	13.2263s	3300	140
lama	34.2953s	2609	131

Table 6: Results of problem 2 configuration 3 with different planners

The results of problem 2 are very different from what we witnessed for problem 1. In fact with the simpler configuration this time all the planners are able, except from *lpg-td* that is the one requiring way more time, to find a solution in the same range of time, with *ff* that requires a little bit more time but is exploring way more states and is producing a longer path. This does not mean that the solution should be discarded a priori. The fact that for this particular problem, planners like *downward, synk* and *lama* can find a solution way faster than the first problem in our opinion is because we added

more constraints on the actions. When the search is done to find the plan, some possible plans that were explored before now are discarded since they don't satisfy one of the added constraints.
The second configuration shows that when augmenting the complexity of the configuration, *ff* and *lpg-td* still are the ones performing worse, with *lpg-td* having the worse result among the five planners both in terms of time and path length. The other three planners are still performing comparably, with *downward* resulting in the faster to solve the problem. The third configuration, as in the previous problem, is the one that drives both *ff* and *lpg-td* to a very long time to find the plan, hence we decided not to consider them. *Downward* and *symk* in this case are performing nearly in the same way, with *symk* needing a little bit more time but in this case, we can say that they are performing equally. On the other side *lama* is taking more than double the time with respect to the other two, even if it is exploring fewer states.

Problem 3

Since problem 3 has a different formulation than the other two, we tested everything differently. The instances of the problems we created starting from the problems of levels 1 and 2 used for the previous problems, but slightly modified to work with what was added to the domain (the equal predicate). Once the problem instance was created, to evaluate the performances of the planner, we first built a simple problem consisting of very simple tasks that we were sure could have been executed (we started from the lower level of task hierarchy). Once we were sure that those tasks could be executed we built the first complete problem consisting of a two-supply delivery with one agent.
The second problem we built was to check if the domain we had built was able to be applied also to more complex scenarios. What we did was insert another agent inside the problem and expand the problem instance to the second level of complexity (as in problems 1 and 2).

Setting	Total time(s)	Num Search Nodes	Path Length
Setting 1	1.102s	44	24
Setting 2	1.718s	119	60

Table 7: Results of problem 3 evaluated with panda on different settings

The execution of the third problem was performed only with the *panda* planner, hence we don't have other planners to compare it to. What we can do is state that augmenting the complexity of the setting we are tackling does not have a higher impact on the performances of the planner as it was happening before, where the change of configuration had greatly increased the value of the time required to plan.

Problem 4

This task has been evaluated using adapted versions of the settings used in problems 1 and 2.

Planner	Total time(s)	Evaluates States	Path Length
Optic	0.83s	162	29
Temporal fast downward	2.27s	1698	31

Table 8: Results of problem 4 setting 1 with different planners

Planner	Total time(s)	Evaluated States	Plan Length
Optic	438.06s	21786	75
Temporal fast downward	49.02s	2215	79

Table 9: Results of problem 4 setting 2 with different planners

In the first configuration, we can see that *Optic* outperforms *Temporal Fast Downward* both in terms of execution time and number of evaluated states. When going from the simple configuration to the more complex one, we can see that this trend is reversed, in fact in the second we can see that *Optic* requires a huge time to find a solution and it is exploring way more states than *TFD*. Despite this, both planners can find a solution even if they may require a lot of time to compute it.

0.2.1 Problem 5

To test the performance of problem 5 we started by setting easy goals for the planner to solve like loading a box with some content and then moving the robot around the map. Once we were sure that the planner was able to solve simple tasks and there existed a correspondence between the times that we described in the actions and the one in the *PlanSys2* fake actions, we proceeded to start testing higher complexity problems. For the first complete problem, we decided to use a simple one-supply delivery just to see if a complete problem could have been fully executed. Then we reached the final test case in which we are using the same level 1 map of complexity that we were using in the previous problem and we decided to set 4 deliveries as the problem's goal. For the whole duration of the execution of the plan, we monitored the correct execution of the actions.

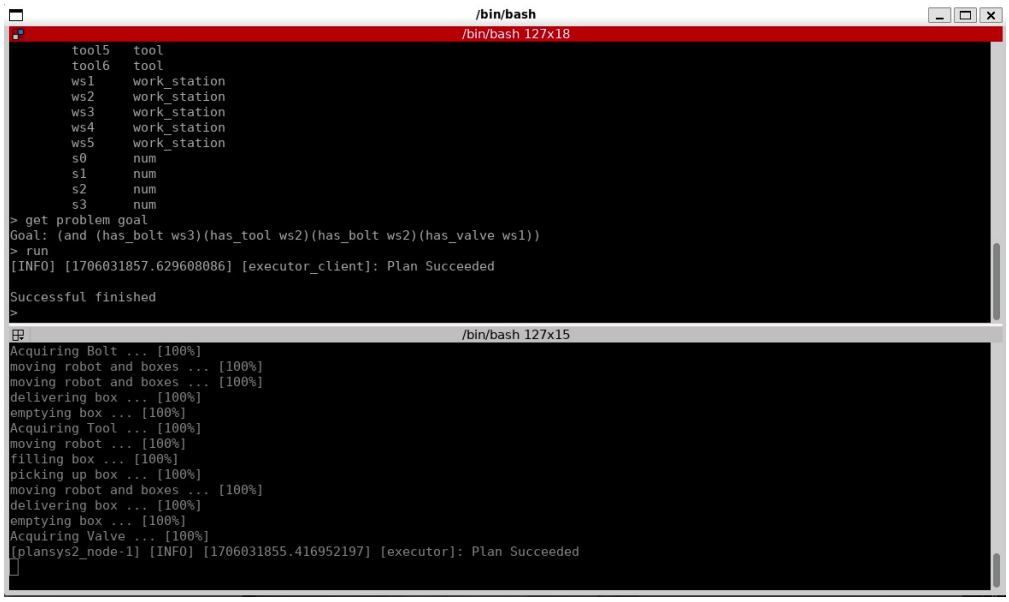

Figure 5: Execution of plan on PlanSys2

As you can see from Fig. 5, the planner can find and execute a solution for the problem we have built.

Planner	Total time(s)	Expanded Nodes	Path Length
Temporal Fast Downward	5.24s	3396	31

Table 10: Execution of PlanSys2 TFD

0.3 Additional information

- The provided directory contains a folder for each of problems 1 up to 4 and a folder named plansys_ws that contains the ROS2 workspace for problem 5
- The instructions to fully run problem 5 can be found in the README of the following GitHub repository
- All the outputs of the planners can be found inside the respective sub-folder named results_problem

Bibliography

[1] plansys2. plansys2_tfd_plan_solver, 2024.