Modeling and solving an emergency services logistics problem

Edoardo Barba (231592)
Mostafa Haggag (229674)

University of Trento

edoardo.Barbar@studenti.unitn.it
mostafa.haggag@studenti.unitn.it

Abstract

This is a report for the project of the Automated planning course taught by Prof. Marco Roveri. We describe the problem provided, the assumptions we made to solve the problem, how we modeled it and the results. We try to compare different planners in terms of efficiency of the number of steps and the fastest to find a solution.

1. Introduction

The problem is an emergency services logistics problem where a number of injured persons are located at known locations and do not move. The objective is to use robotic agents to deliver boxes of emergency supplies to each person. The injured persons are at fixed locations, and the boxes can be filled with different contents such as food, medicine, and tools. The robotic agents can perform actions such as filling, emptying, picking up, moving, and delivering boxes, and can move directly between any location. The goal of this problem is to coordinate the actions of the robotic agents to ensure that the injured persons receive the necessary supplies. We try to model the problem using PDDL and solve it using different planners while testing different initial conditions for the robot and the people. All the code used in here is present in the GitHub [1].

2. Understanding of the problem

We start by modeling the problem where we have different types of objects such as a person, robot, crane, box, location, item, and carrier. The robot needs a carrier to put inside of it the box and it uses the crane to pick up items and pick them down. All the items and the box are always initialized at the base location at the beginning of all of the problems. It is very important to define the location of the box, item, robot, and person so we use a specific predicate for each object to indicate the location. We have in the problem three different items and we use a predicate *is_food*, *is_medicine*, *is_tool* to define the type of the item. In addition, to define the person's requirements we use predicates *need_food*, *need_medicine*, *need_tool*. Furthermore, we have predicates *holding_box*, *holding_item*, and *is_empty_c* to indicate that the crane is used to hold the box, hold the item, and the crane is empty respectively. Finally, we model that the item is inside the box using predicate *inside*. There are more predicates that are added to the problem according to the problem we are solving which we will describe in Section 3.

3. Design Choices

Here are described the design choices used to solve the assignment, and also reported whether generator scripts are used for some part of the assignment (e.g. to generate problem instances).

To solve the assignment, we made both general assumptions that apply to all problems and specific ones that pertain to each problem, as outlined in the subsequent sections. Here are the general assumptions we have made:

- The robots can only move from one location to another, either with or without holding a box, they cannot move holding just an item.
- To insert or remove an object from the box, the robot must place the box on the ground
- In order to deliver an object to a person, the robot has to give the item to the person, removing it from the box
- The robot can use the same box to carry items for different people, so we have no constraints on reusing the boxes, the planner need to find best plan using the available boxes.
- The number of available boxes is not fixed, but it can be determined for each specific problem instance.
- The robot is able to pick up items and place them inside the box, but it is not able to leave items outside of the box at a location, it can only directly give them to people.

With these assumptions being applicable to all problems, we proceeded to establish specific assumptions and design choices for each individual problem.

3.1. Problem 1

In our modeling of problem 1, we made the assumption that the robot is capable of holding only one box at a time and that each box has an unlimited capacity.

We modeled the problem using the following types: *person, robot, crane, box, location* and *item* and the following predicates:

- (*at_b* ?b - box ?l - location): box b is at location l
- (*at_p* ?p - person ?l - location): person p is at location l
- (*at_r* ?r - robot ?l - location): robot r is at location l
- (*at_i* ?i - item ?l - location): item i is at location i
- (*inside* ?i - item ?b - box): item i is inside box b
- (*is_empty_c* ?c - crane): crane c is empty
- (*holding_box* ?c - crane ?b - box): crane c holds box b
- (*holding_item* ?c - crane ?i - item)
- (*is_food* ?i - item): item i is food
- (*is_medicine* ?i - item): item i is medicine
- (*is_tool* ?i - item): item i is a tool

[1] Github link

- (*need_food* ?p - person): person p need food
- (*need_medicine* ?p - person): person p need medicine
- (*need_tool* ?p - person): person p need tool

The domain includes the actions shown in Table 7 of Appendix.

We think that this level of abstraction is enough to model the problem described in the assignment. However it is possible to include more low level actions in order to give more flexibility and control over the robot's actions and increase the scope of the problem. For example an idea could be to add an action that pick an item from a box and an action to put an item in a certain location, so that the robot doesn't necessarily have to give it to a person.

3.2. Problem 2

In this problem, multiple robots are present, each with its own carrier. The capacity of these carriers varies among the robots, allowing for different carrying abilities. Also boxes can contain more than one item and they have a capacity, defined in the problem instance, that is the same for all of them. In order to effectively handle the capacity constraints of both the carriers and boxes, we employed the use of numeric fluents in our approach.

To implement the scenario described, we extended the domain presented in Subsection 3.2 by adding the type *carrier* and adding the predicates described in Table 1 and the following functions:

- (*max_capacity_carrier* ?a - carrier): max capacity of carrier
- (*max_capacity_box*): max capacity of box
- (*box_count* ?c - carrier): number of boxes on carrier c
- (*item_count ?b - box*): number of items in box b

We then remove the actions *move_robot_with_box* and *move_robot_without_box*, substituting them with a unique action *move_robot*. We also substituted actions *pickup_box* and *pickdown_box*, with *load_box* and *unload_box*, to load and unload the box to the carrier.

Predicate name	Parameters	Description
belongs_carrier	?a - carrier ?r - robot	carrier c belongs to robot r
on	?b - box ?a - carrier	box b is on carrier c

Table 1

3.3. Problem 3

The hierarchical domain description language [4] is an extension of PDDL 2.1 [2]. The main task in this part of the problem is to leverage the scenario of Subsection 3.2 while addressing the problem with hierarchical task networks. We use the same actions introduced in problem 2 but we propose a new list of tasks and methods to model the problem. We used open source HTN planner called PANDAS developed by the university of ULM.
Due to the fact that PANDAS does not support numeric fluents, we did an assumption that each robot can put only 1 box inside the carrier for all robots and that the maximum capacity of the box is just one. This makes the problem similar to problem 3.1.

We have primitive tasks and methods that directly call an action. We are using the same actions as used previously but with the addition of 1 action called noop that indicates that the robot is already at the current position.
As for the non-primitive tasks, firstly we have *deliver_box* called by method *m_deliver_box* that controls the robot to go to a box, pick up a box and go to the destination location of the box and unload it. *catch_item_put_in_box*, *deliver_food*, *deliver_tool* and *deliver_medicine*. Secondly, we have *catch_item_put_in_box* called by method *m_catch_item_put_in_box* that controls the tasks of picking up from location and putting down into the box. We have another method for the same task called *m_catch_item_put_in_box_2* to allow to move the box to the location of the item before loading the item inside of it. Lastly, we use a specific task *deliver_food* with method *m_deliver_food* to define that person needs food and the robot has to catch it into the box, deliver the box, pick it up and put down the food. We have similar tasks and methods for delivering tools and medicine.

3.4. Problem 4

We leverage from the problem 3.2 and we try to introduce durative actions. OPTIC [1] is a temporal planner for use in problems where plan cost is determined by time-dependent goal collection costs and we use it during our problem. We choose arbitrary actions time for different actions and we have the possibility of having actions that can be executed in parallel if it makes sense. We mainly restrict the robot from doing any durative actions in our problem but we allow durative actions when having more than 1 instance of the robot. We mainly see when introducing 2 robots a cooperation between both to optimize the time. We introduce 3 new predicates called *satisfied_p_for_food satisfied_p_for_food satisfied_p_for_food* as they are used to indicate the goal because we were not allowed to use negation during the definition of the goal by the planner. We have added Table.8 in Section 7 that describes each action in this problem.

3.5. Problem 5

We implement problem 4 within Plansys2 [6] in Ros 2 Humble distribution [5] using fake actions. We create a package called **plansys2_problem5** and we implement in C++ the different actions that can be used inside the PDDL file. We create a launch file in python for all different nodes. We assume that we have 2 robots in the environment so we add 2 instances for each action and adding more robots would require modifications in the launch file and in the CMake file. There were not any new assumptions made in this problem. We can see in Table.2 all the parameters set in the launch file for the different actions.

Actions	Time (ms)	Increment progress
move_robot	400	0.01
load_box	400	0.1
unload_box	400	0.1
pickup_item_from_location	400	0.17
put_item_in_box	400	0.17
pick_item_from_box	400	0.17
pickdown_food	400	0.17
pickdown_tool	400	0.17
pickdown_medicine	400	0.17

Table 2: *We see the time for each action set in the launch file.*

4. Results

We describe here in detail the different settings that we tested for each problem and we try to compare them with different planners and see the difference in performance. All the problems that we ran and tested are added to the GitHub link with the commands used for the results we obtained. The GitHub repository is organized in a way that we have each source and problem file inside a folder with the problem name. A markdown is there to show all the different files inside each folder and detailed commands on how to run each problem.

4.1. Problem 1

We conducted experiments using various search strategies on different problem scenarios in the domain.
The first scenario involves a robot located in the same location of a box and 3 items: food, tool, and medicine, while a person in need of all three items is located elsewhere. In the second scenario, we added 3 items, and a person located elsewhere who also requires three items.

We show the results of the following planners: A star search with goal count heuristic, fast-forward, lama and lama-first. The goal count heuristic is a simple and computationally efficient heuristic that estimates the cost of reaching a goal state from a current state by counting the number of predicates in the goal state that are not yet satisfied in the current state.

It is a not admissible heuristic, and as a result, it may not always provide an accurate estimate of the optimal plan because we must often unachieve individual goal literals to get closer to the goal.

Fast-forward planner relies on forward search in the state space, guided by a heuristic that estimates goal distances by ignoring delete lists [3].

LAMA is a more sophisticated planner that uses a multi-heuristic search to find an optimal solution. It uses of a pseudo-heuristic derived from landmarks, propositional formulas that must be true in every solution of a planning task, and it combines it with a variant of the previously mentioned FF heuristic. Both heuristics are cost-sensitive, focusing on high-quality solutions in the case where actions have non-uniform cost (which is not the actual case). [8].

LAMA-First is a simpler planner that uses a single heuristic function to find a good solution quickly and does not necessarily guarantee the optimality of the solution.

The Lama planner is unique among these planners in that it guarantees optimality. This means that the plan length found by the Lama planner represents the optimal plan length for the given problem and it is therefore important for understanding the true optimal solution for a given problem.
We invoke Lama, ff and Lama-first using *planutils* [7] and Astar with goal count using *downward* [3].

In Table 3 we can see results of some planners we tried to use.

For the first problem instance, we can see from the table that all planners take very short time to find the solution and all planners except lama-first, find the optimal solution.
Looking at the second problem instance, instead, we can see that lama-first finds a solution much faster than the others although it a very long plan compared to the optimal one. Also FF finds the plan in short time, finding also a very good solution, even tough it is not optimal. It appears that the A* algorithm with the goal count heuristic is the most effective strategy among the planners we tested. However, it is important to note that this conclusion is likely influenced by the relatively simple state space of the problem and the proximity of the goal state to the initial state. In such cases, the goal count heuristic proves to be a reliable estimate of the remaining distance to the goal. In addiction goal count heuristic is not admissible so, in general it will not find the optimal solution.

Prob instance	Planner	Search Time (s)	Plan Lenght
1	*goal-count*	0.15	15
	ff	0.01	15
	lama-first	0.01	27
	lama	0.02	15
2	*goal-count*	10.71	27
	ff	2.75	31
	lama-first	0.03	53
	lama	108.03	27

Table 3

Prob instance	Planner	Search Time (s)	Plan Lenght
1	*sat-hadd*	1,39	22
	opt-blind	2,56	20
	opt-hmax	13,0	20
2	*sat-hadd*	0,6	32
	opt-blind	25,42	30
	opt-hmax	143,17	30

Table 4

4.2. Problem 2

In this second task, we also conducted experiments by applying various planners to a range of different problems. To support the numeric fluents we selected ENHSP (Expressive Numeric Heuristic Search Planner) [10] as our planner system, as it is equipped with the capability to handle this requirement.
The initial problem we examined (instance 1) consisted of 2 distinct robots, 3 boxes and 2 people. A robot is equipped with a carrier with capacity of 1 and the other with a capacity of 2. We fixed the capacity of the boxes to 1 and set that a person need only 1 item while the other needs 2 items.

In the second problem instance, we added 1 item and 1 person on the same location of another one.

We tried different planners and the results are showed in table 4.

We used a baseline planner (opt-blind) that uses A* with a simple blind heuristic which always returns a value of 0, regardless of the current state of the search. It is an admissible heuristic, so used in combination with A* will return an optimal solution.

Another planner we tried is sat-hadd which uses a Greedy Best First Search with numeric additive heuristic, which is based on the idea that the cost of reaching the goal is the sum of the costs of achieving each individual goal literal [9].

The last planner we used is opt-hmax, based on A* search with hmax numeric heuristic which estimates the cost of reaching the goal by taking the maximum cost of reaching the goal from any single action. It is an admissible heuristic so this planner guarantees optimality [9].
It appears that in the first problem instance, the sat-hadd and opt-blind planners were able to quickly find a solution, with sat-hadd being slightly faster. However, it should be noted that sat-hadd did not find the optimal solution. On the other hand,

opt-hmax took significantly longer to find a plan.

In the second problem instance, sat-hadd was able to find a good plan in a very short amount of time, significantly faster than opt-blind and opt-hmax.

It appears that sat-hadd is well-suited for this particular problem, able to find a good plan quickly, whereas opt-blind and opt-hmax take much longer.

Despite its simplicity, opt-blind seems to work better than opt-hmax in this problem instances, finding a solution in less time.

4.3. Problem 3

We conduct experiments on 2 different problems. The first problem contains 1 robot with 1 person that requires tools and medicine while in the second problem, we introduce 2 robots and 1 person that requires tools and medicine. We mainly compare different settings in terms of checking the possibility to define a hierarchy for the goal executed or without any hierarchy.

We noticed that putting a hierarchy for the goal helps the planner to be able to reach the optimal solution faster. The planner was able to reach the same plan with the same number of steps for both cases with and without a ranking for the goals. We also noticed that when trying to make the problem bigger with more robots and people it takes a lot of time for the planner to reach a solution in contrast with the planner used in Subsection 4.2

Example	Search Time (ms)	Cost of solution
example 1 ordered goal	18	18
example 1 unordered goal	4267	18
example 2 ordered goal	29	18
example 2 unordered goal	217	18

Table 5: *In this table we can see the results of Pandas planner for different example files.*

4.4. Problem4

We experiment with 2 different scenarios with 2 robots. The main difference between the 2 robots is that the maximum carrying capacity for robot 1 is 2 while the maximum carrying capacity for robot 2 is 1. We set the maximum capacity per box equal to 1. In the first experiment, we have 2 persons at different locations with the first person requiring food and the second requiring tool. In the second experiment, we have 2 persons at the same locations with the first person requiring food and a tool and the second requiring a tool. We experiment with 3 different settings for OPTIC [1] where the main difference between the commands is that we weigh the A^* algorithm by weight and the third command allows us to go to the best first search.

We have added all the different commands used to the README file in GitHub.

Example	Search time (sec)	Cost (sec)	States evaluated
example 1 command 1	1.78	172	6546
example 1 command 2	5.57	72.04	19390
example 1 command 3	5.64	72.04	19310
example 2 command 1	35.1	304.007	94697
example 2 command 2	350.18	94.010	694333
example 2 command 3	385.60	94.010	694290

Table 6: *In this table we can see the results of OPTIC for different example files and different commands.*

As seen in Table. 6, we can see that weighting the A^* helped us to achieve a better solution in terms of the cost but lead to more states required to be evaluated in both problems. By looking at the solutions provided by running command 2 and command 3 on the different problem instances, we can say that we reached the most optimized solution for the problem where each robot is using its maximum carrying capacity and the robots are cooperating together to do actions that allow us to get the most optimized solution.

4.5. Problem5

For problem 5, we created very similar examples to the ones created for Subsection 4.4. we created several nodes for the 2 instances of the robot and we added pictures for the results of fig.1 and fig.2.

5. Conclusion

In conclusion, this report has described an emergency services logistics problem and the modeling and solving of the problem using PDDL and various planners.

We modeled the subproblems by making some general and specific assumptions

we love you so roveri so much we want to do thesis with you see u, bye ¡3

6. References

[1] BENTON, J., COLES, A., AND COLES, A. Temporal planning with preferences and time-dependent continuous costs. *Proceedings of the International Conference on Automated Planning and Scheduling 22*, 1 (May 2012), 2–10.

[2] FOX, M., AND LONG, D. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research 20* (dec 2003), 61–124.

[3] HOFFMANN, J. Ff: The fast-forward planning system. *AI Magazine 22*, 3 (Sep. 2001), 57.

[4] HÖLLER, D., BEHNKE, G., BERCHER, P., BIUNDO, S., FIORINO, H., PELLIER, D., AND ALFORD, R. Hddl: An extension to pddl for expressing hierarchical planning problems. *Proceedings of the AAAI Conference on Artificial Intelligence 34*, 06 (Apr. 2020), 9883–9891.

[5] MACENSKI, S., FOOTE, T., GERKEY, B., LALANCETTE, C., AND WOODALL, W. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics 7*, 66 (2022), eabm6074.

[6] MARTÍN, F., GINÉS, J., MATELLÁN, V., AND RODRÍGUEZ, F. J. Plansys2: A planning system framework for ros2, 2021.

[7] MUISE, C., POMMERENING, F., SEIPP, J., AND KATZ, M. Planutils: Bringing planning to the masses. In *32nd International Conference on Automated Planning and Scheduling, System Demonstrations and Exhibits* (2022).

[8] RICHTER, S., AND WESTPHAL, M. The LAMA planner: Guiding cost-based anytime planning with landmarks. *CoRR abs/1401.3839* (2014).

[9] SCALA, E., HASLUM, P., AND THIÉBAUX, S. Heuristics for numeric planning via subgoaling. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence* (2016), IJCAI'16, AAAI Press, p. 3228–3234.

[10] SCALA, E., HASLUM, P., THIEBAUX, S., AND RAMIREZ, M. Interval-based relaxation for general numeric planning. In *Proceedings of the Twenty-Second European Conference on Artificial Intelligence* (NLD, 2016), ECAI'16, IOS Press, p. 655–663.

7. Appendix

7.1. Extra material for problem 1

Action name	Description
move_robot_with_box	moves a robot from one location to another while holding a box
move_robot_without_box	moves a robot from one location to another without holding a box
pickup_box	picks up a box
pickdown_box	picks down a box
pickup_item_from_location	picks up an item from a location
put_item_in_box	puts an item in a box
give_food	gives food to a person
give_medicine	gives medicine to a person
give_tool	gives a tool to a person,

Table 7

7.2. Extra material for problem 4

Actions	Duration	Conditions	Effects
move_robot	50	At the start, the robot is in a specific location X. Overall the action, the crane is empty and the crane belongs to the robot.	At start of the action the robot is longer at location X and at end the robot is at location Y.
load_box	5	At the start, the box is at a certain location and the crane is empty and the robot box count does not exceed the maximum a number of the carrier capacity. Overall the action the crane and carrier belongs to the robot and the robot is at location X.	At the start, the crane is not empty and the box is not at location and we increase the box count by 1. In the end, the box is on the carrier, and the crane is empty.
unload_box	5	At the start, the crane is empty and the box is in the carrier. Overall the action, crane, and carrier belong to the robot and robot at location X.	At the start, the box is not in the carrier and the crane is not empty and decrease carrier box count by 1. In the end, the box is at the location and the crane is empty.
pickup_item_from_location	3	At the start, the crane is empty and the item is at the location. Overall the action, robot at the location, and crane belong to the robot.	At the start, the crane is not empty and the item is not at location. In the end, crane is holding the item.
put_item_in_box	3	At the start, the crane is holding the item and the item count is less than the box count max capacity. Overall the action, the robot is at the location, and box at the location and crane belong to the robot.	At the start, the robot is not holding items by the crane and we increase the box count, In the end, the item is inside the box, the crane is empty.
pick_item_from_box	3	At the start, the crane is empty and the item is inside the box. Overall the action, crane belongs to the carrier, and the robot and the box are at location X.	At the start, the item is not inside the box and the crane is not empty and we decrease the box count. In the end, the crane holds the item.
pickdown_food	3	At the start, the person needs food and the crane is holding the item. Overall the action, the item is food , person and robot are at location X.	At the start, the crane is not holding an item. In the end, the person does not need food and the crane became empty and the person is satisfied with the food.
pickdown_tool	3	At the start, the person needs a tool and the crane is holding the item. Overall the action, the item is a tool, person and robot are at location X.	At the start, the crane is not holding an item. In the end, the person does not need tool and the crane became empty and the person is satisfied with the tool.
pickdown_medicine	3	At the start, the person needs medicine and the crane is holding the item. Overall the action, the item is medicine, person and robot are at location X.	At the start, the crane is not holding an item. In the end, the person does not need medicine and the crane became empty and the person is satisfied with the medicine.

Table 8: *In this table, I am explaining all the actions used for problem 4.*

7.3. Extra material for problem 5

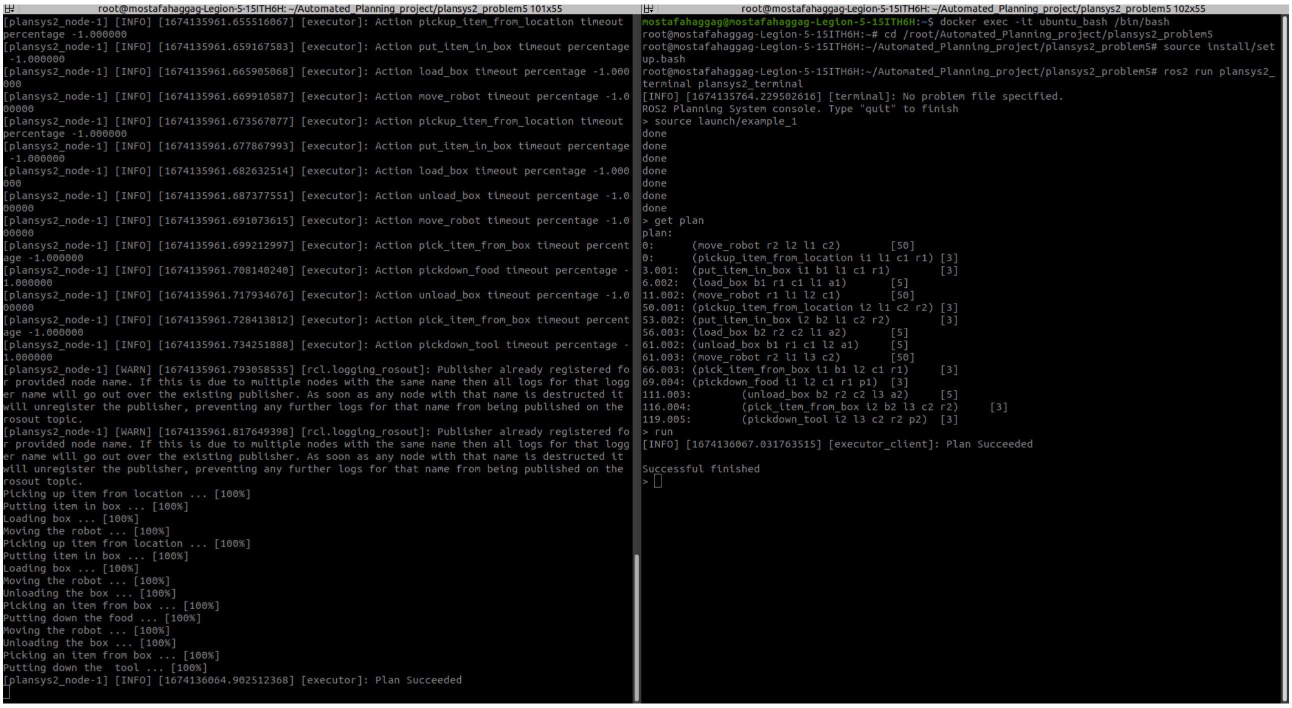

Figure 1: *A figure showing the results of example 1 ran in Plansys2 [6])*

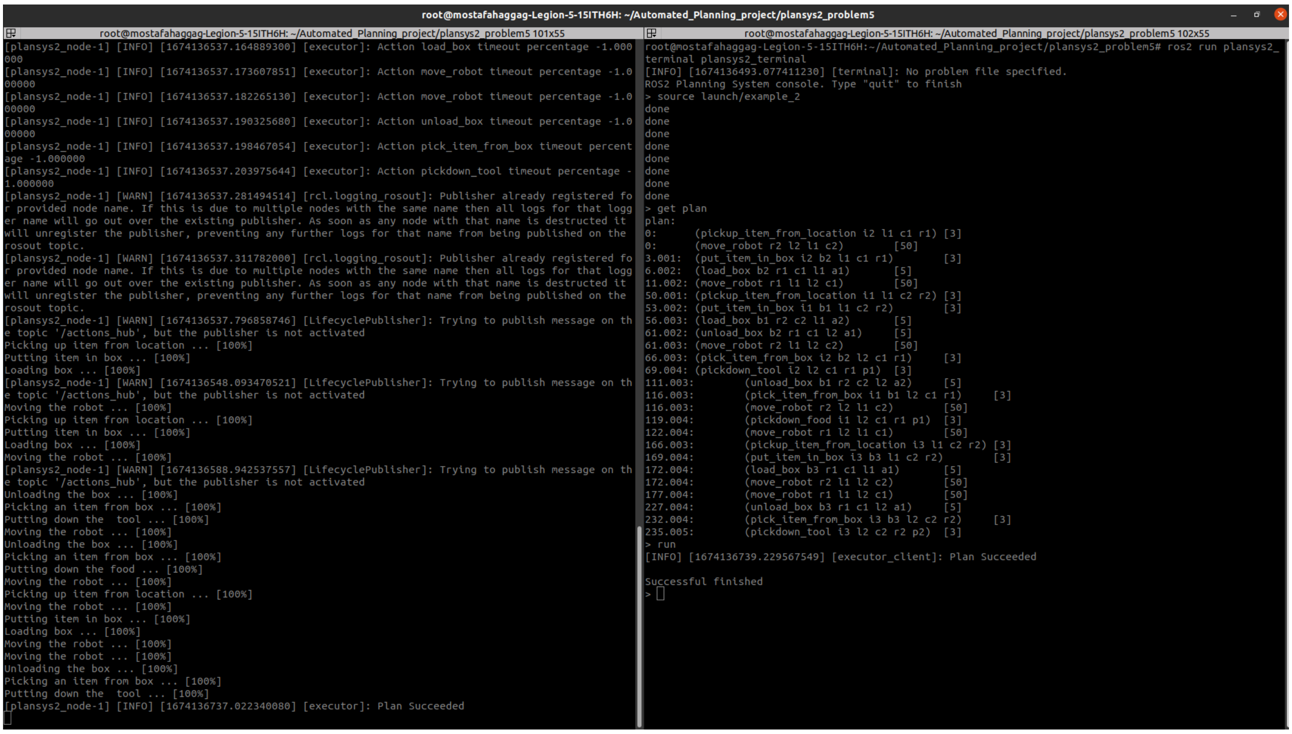

Figure 2: *A figure showing the results of example 2 ran in Plansys2 [6])*