EMERGENCY SERVICES LOGISTICS
AUTOMATED PLANNING PROJECT - ACADEMIC YEAR 2022-2023

Carlo Marotta
carlo.marotta@studenti.unitn.it
Master in Artificial Intelligence Systems - University of Trento

ABSTRACT

The report is about a project on Automated Planning[1] completed as part of a Master's program in Artificial Intelligence Systems at the University of Trento. The project explored the challenges of automated planning in dynamic and resource-constrained environments for real-world applications such as robotic agents delivering essential contents during emergency situations. PDDL [1] and PlanSys2 were used to address five key problems, and the report covers problem description, assumptions, design choices, modeling, organization, and results, including a comparison of different planners' efficiency. The project serves as a proof of concept for future applications and establishes a foundation for further research and development.

1. INTRODUCTION

This report explores the application of automated planning techniques in solving a real-world problem concerning emergency services logistics optimization. The problem involves delivering emergency supplies to injured individuals at fixed locations using autonomous robotic agents, and we use offline planning techniques, which entail creating a plan before executing tasks. We begin by analyzing the problem step-by-step, starting from a basic version in Problem 1 and gradually increasing in complexity as we progress towards Problem 5. To achieve this, we utilize different planners such as *Downward*, *PANDA*, *Optic*, and *PlanSys2*, which support various planning languages and techniques.
The report is structured as follows: in **Section 2**, we provide our interpretation of the problem for all five problem statements. **Section 3** focuses on explaining the design choices we made and the characteristics of the problem. Moving on to **Section 4**, we present the results obtained from our planning process, accompanied by detailed comments on those results. In **Section 5**, we delve into the contents of the archive and discuss how it has been organized. Finally, in **Section 6**, we conclude the report with our thoughts and findings, followed by a discussion on potential future implementations that could further enhance this project.

[1]All the project-related documentation is available in the GitHub repository link.

2. UNDERSTANDING OF THE PROBLEM

The assignment involves the design of an emergency logistics service that utilizes robotic agents to deliver essential contents to injured individuals. To accomplish this, we must create a planning system that effectively manages the activities of these agents as they transport boxes filled with emergency supplies to predetermined locations. Several assumptions underlie the problem scenario, including:

- **Injured person**: each injured person is located at a fixed, predetermined location, which does not change. Furthermore, each individual may or may not have a supply box containing various contents, which may differ from person to person. Some individuals may require one or more of the supplies, while others may require none at all. It's also possible for multiple injured individuals to be located at the same location.

- **Box**: each box is initially located at the depot and can be filled with a variety of emergency contents. It's necessary to keep track of the contents of each box in a generic manner so that new types of supplies can be added in the future.

- **Content**: these are transported inside the boxes and can be of three types - food, medicine, and tools. The depot serves as the starting location for all of these contents.

- **Robotic agent**: the robotic agents can fill an empty box with a specific content provided that the box, agent, and content are all at the same location. Additionally, the agents can unload a box by leaving its content at the current location, which results in all people there receiving the supplies. They can also pick up a single box and load it onto themselves if it is at the same location, move to another location while carrying the box (if it has been loaded), or deliver a box to a specific person at the same location.

- Location graph: the graph of locations is fully connected, which means that the robotic agents can move directly between any two locations without following a specific route.

Starting from problem 2, the robotic agents have been granted additional capabilities, which include:

- **Carrier**: the agents are now equipped with a carrier that enables them to transport multiple contents together. The robotic agents can load up to four boxes onto a carrier provided that they are all at the same location. Once loaded, the carrier can be transported to a location where supplies are needed, and the agent can unload one or more boxes at that location. The carrier can then be moved to other locations to unload additional boxes, without returning to the depot until all boxes on the carrier have been delivered.

- **Carrier capacity**: the problem file should specify the capacity of the carriers. It's important to keep track of the number of boxes on each carrier and the total number of boxes for each robotic agent to prevent overloading.

In summary, the emergency logistics service is designed to efficiently deliver essential contents to injured people at fixed locations using robotic agents and carriers. The planning system must manage the agents' activities, keep track of the contents required by each injured individual, and ensure efficient delivery of the supplies.

3. DESIGN CHOICES

In this section, we will describe the implementation choices made for each of the 5 problems tackled. Specifically, each problem contains two subfolders representing two different solutions: one generally more "simple" and one more complex using "fluents" (except for problem 1 and problem 3).
In general, the implemented choices are similar to each other and increase in complexity from the first problem to the fifth, in which all the assumptions listed in the previous section have been respected. Additionally, the following initial state conditions are required in all problems:

- all boxes are located in a position that we call **depot**;

- all contents to be delivered are located at the depot;

- there are no injured people at the depot;

- the robotic agent is located at the depot.

The objective is also common to all exercises, in particular, some people have specific contents (e.g. medicine, food, tools), some people may not need food, medicine, or tools, and finally, some people may need both food and medicine, or food and tools, or all three, and so on.

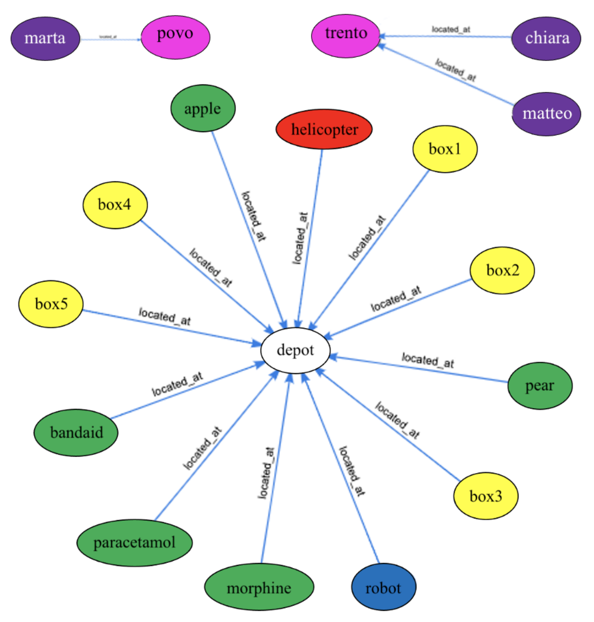


Fig. 1: Initial state of our problems.

3.1. Problem 1

The first problem is of the lowest complexity, wherein the robotic agent is required to load one item at a time, transport it to a location where an injured person is in need, and then return to the depot to repeat the same process until all injured people have been attended to. In the overall scenario of the first problem, the following types are shared by both solutions:

- `location`: represents a location in the world;

- `movable`: represents an object that can be moved and/or is located at a location;

- `injured_person`: represents a person who is injured and needs medical attention;

- `robotic_agent`: represents an autonomous robot that can perform tasks;

- `box`: represents a container for contents;

- `content`: represents the contents that can be transported in a box and delivered to an injured person;

- `food`, `medicine`, `tool`: represent the different types of content that can be delivered to an injured person.

Moreover, has been defined a constant `depot`, that represents the location where all objects start at. In the context of the problem, the objects (e.g., boxes, contents, robot) are initially located at the depot before being moved to their respective destinations. Also the following predicates that describe various conditions and relationships between objects and their properties in a simulated world are common in both subfolders:

- `(located_at ?m - movable ?l - location)`: whether a movable object is located at a particular location in the world.
- `(is_empty ?b - box)`: whether a box is empty or not.
- `(has_content ?b - box ?c - content)`: whether a box has a certain type of content.
- `(is_loaded ?r - robotic_agent ?b - box)`: whether a robot has a box loaded with content.
- `(is_unloaded ?r - robotic_agent)`: whether a robot is not currently loaded with a box.
- `(need_food ?p - injured_person)`: whether an injured person needs food.
- `(need_medicine ?p - injured_person)`: whether an injured person needs medicine.
- `(need_tool ?p - injured_person)`: whether an injured person needs a tool.

These predicates are used to define the initial state and the goal state of a problem, as well as the actions that can be taken in the world to transform one state to another. The common actions are as follows:

- `move_robot_with_content`: moves the robotic agent from the depot to another location while carrying a filled box of specified content, provided that the agent and box are both currently located at the depot and the agent is currently loaded with the box and the box contains the specified content.
- `deliver_food/medicine/tool`: delivers a content to an injured person at a specified location, provided that the person, robotic agent, and box with the content are all located at that location, the agent is currently loaded with the box, and the person needs food/medicine/tool.
- `move_robot_without_content`: moves the robotic agent from a specified location to the depot while carrying an empty box, provided that the agent and box are both currently located at the specified location, the agent is currently loaded with the box, and the box is currently empty.

3.1.1. *Simple*

In the simplest scenario, we have implemented the following actions to ensure the proper execution of the goals:

- `fill_box`: fills a box with a specified content when the robotic agent, box, and content are all located at the depot and the box is currently empty and not already loaded onto the robotic agent.
- `load_robot`: loads a filled box onto the robotic agent when the agent and box are both located at the depot and the agent is currently unloaded.
- `unload_robot`: unloads an empty box from the robotic agent, provided that the agent and box are both located at the depot, the agent is currently loaded with the box, and the box is currently empty.

Overall, the hierarchy of types (defined in the domain file) and objects (defined in the problem file) is structured as follows:

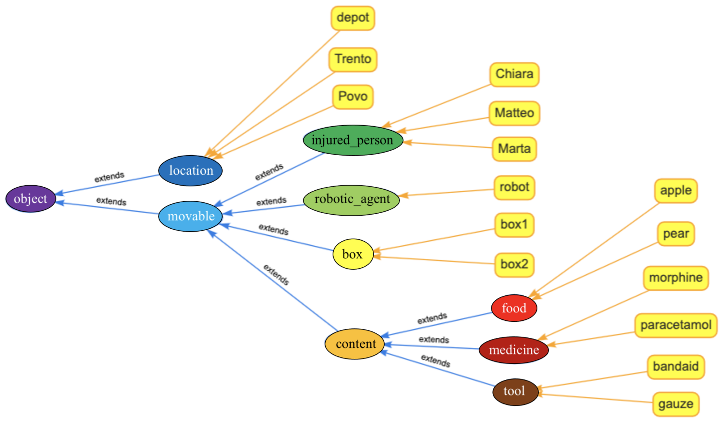


Fig. 2: Hierarchy of types and objects of our problems.

3.1.2. *Crane*

In the complex scenario, we have added a crane always located at the depot that helps the robotic agent to pick up/down boxes and contents. According to this, we have added the following predicates:

- `(holding_box ?g - crane ?b - box)`: whether a crane is currently holding a box;
- `(holding_content ?g - crane ?c - content)`: whether a crane is currently holding a content;
- `(is_free ?g - crane)`: whether a crane is currently not holding any object (neither a box nor a content);
- `(is_delivered ?c - content)`: whether a content has been successfully delivered to an injured person.

The following actions are related to a scenario where a crane is used to move boxes and contents in a depot to be delivered to injured people:

- `crane_pick_up_content`: a crane picks up a content from the depot, given that it is free and there is an empty box available;
- `crane_fill_box`: a crane fills an empty box with a content. The box must be at the depot, and not already loaded on a robot;
- `crane_pick_up_box`: a crane picks up a filled box, that must be at the depot, to be loaded onto a robot for delivery, and there must be an available unloaded robot;
- `crane_load_robot`: a crane loads a filled box onto an available unloaded robot, that must be at the depot and not already loaded;
- `unfill_box`: a robot unloads a content from a box, given that it is loaded and the robot is at a specific location;
- `crane_unload_robot`: a crane unloads a box from a loaded robot at the depot, given that the box is empty and the crane is free;
- `crane_put_down_box`: a crane puts down an empty box at the depot, given that it is holding the box and the box is empty.

3.2. Problem 2

In the second problem, as already anticipated at the end of section 2, we exploit the scenario of the simple scenario of the first problem with the various extensions in order to consider an alternative way of moving the boxes (for example, trucks or helicopters or drones). The initial conditions and goal are the same, except that the robotic agent is initially empty and its capacity must be set to 4.

3.2.1. Simple

For the simple scenario, we have incorporated two types based on the extensions that were requested:

- `carrier`: an object used for transporting boxes;
- `compartment`: a space within a carrier that can contain a box.

The following two predicates have also been added for proper use of space on the carrier:

- `(is_free ?s - compartment)`: whether the compartment is currently unoccupied by a box;
- `(is_occupied ?s - compartment ?b - box)`: whether the compartment is currently occupied by the specified box.

Additionally, changes have been made to the following actions to comply with the new conditions:

- `load_carrier`: loads a filled box onto a carrier, provided that there is enough space on the carrier to load the box. In this case, we are keeping track of which boxes are on each carrier and how many boxes there are in total on each carrier, so that carriers cannot be overloaded;
- `move_carrier_with_box`: moves a robotic carrier with a filled box from one location to another, provided that there is at least one compartment on the carrier that is not empty;
- `move_carrier_without_box`: moves a robotic carrier from one location to the depot without any boxes such that does not have to return to the depot until after all boxes on the carrier have been delivered.

In this scenario, the arrangement of types and objects follows the same hierarchy as depicted in Fig. 2. However, there are also some additional components included:

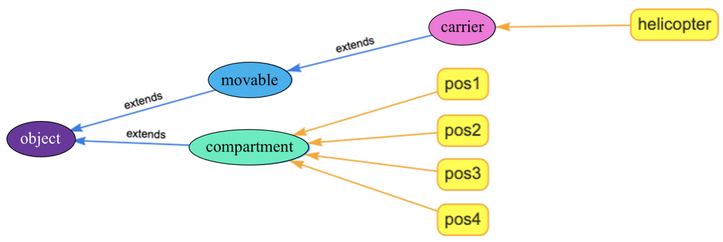
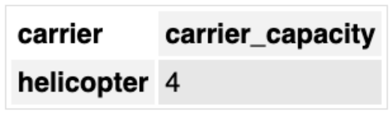


Fig. 3: Additional types and objects for next scenarios.

3.2.2. Fluents

In the second scenario, fluents were utilized to specify the capacity of the carrier, and to keep track of the number of boxes on each carrier, as well as the total number of boxes on each carrier. Instead of defining the type `compartment`, we have used the function `(carrier_capacity ?a - carrier)`, which is fixes to 4 in the problem file as per the assignment requirements:



carrier	carrier_capacity
helicopter	4

Fig. 4: Definition of carrier capacity.

Additionally, the previously described actions are adapted so that the capacity of the carrier decreases or increases depending on whether a box was loaded or unloaded, respectively.

3.3. Problem 3

In Problem 3, the main task is to rewrite the code using the Hierarchical Domain Description Language (**HDDL**) [2], that is an extension of PDDL 2.1 [3]. Here, we utilize the simple scenario of Problem 2 as a basis, but this time we employ hierarchical task networks (HTN) [4] to solve the problem. We maintain the same set of actions as the solution proposed for problem2, along with the same initial condition and goal. However, we introduce the use of `:tasks` and `:methods` to help us achieve the objective using HTN. In both scenarios we defined the following `:tasks`:

- `delivery_1_content`: a task to deliver 1 content to an injured person;
- `delivery_2_contents`: a task to deliver 2 contents to an injured person;.
- `delivery_3_contents`: a task to deliver 3 contents to an injured person;.
- `move`: a task to move from a location to another one;
- `prepare_content`: a task to prepare a content for delivery;
- `prepare_box`: a task to prepare a box for delivery;
- `prepare_carrier`: a task to prepare a carrier for delivery;
- `prepare_delivery`: a task to prepare the delivery;
- `final_delivery`: a task to perform the final delivery to an injured person.

The following are the common `:methods` for both subfolders:

- `m_delivery_1_content_in_progress`: delivers 1 box with one content to an injured person such that the carrier is prepared before moving/checking the robot's position, the robot's position is checked/moved before preparing the delivery, and the delivery is prepared before moving/checking the robot's position again;
- `m_delivery_2_contents_in_progress`: delivers 2 boxes with two contents to an injured person such that the carriers are prepared before moving/checking the robot's position, the robot's position is checked/moved before preparing the deliveries, and the deliveries are prepared before moving/checking the robot's position again;
- `m_delivery_3_contents_in_progress`: delivers 3 boxes with three contents to an injured person with the same order of the previous methods;
- `m_complete_carrier_preparation`: prepares the carrier with a box and its content such that the content of the box is prepared before loading the box onto the carrier;
- `m_complete_content_preparation`: fill the box with the content at a specific location;
- `m_complete_box_preparation`: loads a box onto a carrier;
- `m_complete_movement_from_depot`: moves a loaded robot from the depot to a specified location;
- `m_complete_movement_to_depot`: moves an unloaded robot from a specified location to the depot;
- `m_complete_delivery_preparation`: prepares the delivery by unloading the box from the carrier at a specified location and then completing the final delivery to an injured person;
- `m_complete_food_delivery`: food delivery;
- `m_complete_medicine_delivery`: medicine delivery;
- `m_complete_tool_delivery`: tool delivery.

3.3.1. Htn1

In this scenario, we defined as goal in the problem file to complete three deliveries such that the first delivery must be completed before the second delivery and the second delivery must be completed before the third delivery.

3.3.2. Htn2

In this more complex scenario[2], we defined another method `m_remain_here` used to call the action `check_location` to ensure that the robot does not move when it does not need to. Also in this case the goal is to complete three deliveries with the difference that in this case the first two do not have a hierarchical order and must precede the third delivery.

3.4. Problem 4

In this task, we have to expand on the scenarios presented in problem 2 by incorporating durative actions with varying durations. Additionally, we will consider the feasibility of executing actions in parallel where appropriate, while keeping in mind realistic constraints, therefore we added the predicate `(ready ?r - robotic_agent)` that determines whether a robotic agent is ready for performing a task. The initial conditions and goals remain the same as those presented in previous tasks. To solve this problem, we utilized the OPTIC

[2]The planners for HDDL do not support numeric fluents.

planner [5] which is designed for temporal planning where the cost of the plan is determined by preferences or time-dependent goal-collection costs. We modified the actions (domain file) and the goal (problem file) to comply with its limitation[3], by adding the following predicates:

- `(has_food ?p - injured_person)`:
  whether an injured person has received food;
- `(has_medicine ?p - injured_person)`:
  whether an injured person has received medicine;
- `(has_tool ?p - injured_person)`:
  whether an injured person has received a tool.

3.4.1. Simple

An extra action, named `unfill_box`, has been included in this scenario in addition to the modifications mentioned before. Its purpose is to remove the specified content from the box to allow for the subsequent delivery to the injured person.

3.4.2. Fluents

In the case of fluents, to ensure the correct functioning of the Optic planner without using negative preconditions, an additional function called `(max_capacity ?a - carrier)` has been introduced. It defines the maximum capacity available on the carrier and remains constant throughout the planning process.

3.5. Problem 5

In this subsection, we present our implementation of problem 5 using the PlanSys2 [6] framework in ROS2 Humble distribution [7]. Our implementation extends problem 4, in which the objective of problem 5 is to simulate the execution of actions by associating C++ codes with each action defined in the PDDL file. We created two packages named **plansys2_problem5_simple** and **plansys2_problem5_fluents** for the "simple" and the "fluents" scenario, respectively, and defined the required dependencies in the CMake files. The packages include different nodes for each action, then we have implemented a set of C++ codes for each action, which are executed as "fake actions" with a predefined duration of 200ms, and we also defined the necessary launch files and PDDL files required for the planning process. We tested our implementation by calling the APIs and simulating the execution of actions, without having access to a physical or virtual robot for testing, so we relied on the "fake actions" to simulate the actions.

[3]The OPTIC planner has limited support for ADL and does not support negative preconditions, except in the rules for derived predicates.

4. RESULTS

In order to thoroughly evaluate the performance of different planners in solving the defined problems, we conducted extensive testing and analysis. This section provides a detailed overview of the settings we tested for each problem and compares the results obtained by various planners. All results are saved in log format within a subfolder named **results** for each scenario of each problem. Additionally, other results are presented as screenshots in the *README.md* file of the GitHub repository mentioned on page 1.

4.1. Problem 1

For our experiments in the first problem, we utilized several planners, including:

- **FF** (Fast-Forward) [8]: it uses forward search in the state space and a heuristic that estimates goal distances without considering delete lists;
- **LAMA** [9]: it employs a multi-heuristic search to find an optimal solution. It combines a pseudo-heuristic based on landmarks and propositional formulas, which must be true in every planning task solution, with a modified version of the FF heuristic. These heuristics take into account non-uniform action costs and prioritize high-quality solutions;
- **LAMA-FIRST**: it is a simpler version that quickly finds a satisfactory solution without guaranteeing optimality. It uses a single heuristic function;
- **A*** search with the **goal count** heuristic: it efficiently estimates the cost of reaching a goal state by counting unsatisfied predicates. However, it may not be admissible or accurately estimate the optimal plan due to the frequent need to undo achieved goal literals for progress towards the overall goal.

To invoke the FF, LAMA, and LAMA-FIRST planners, we utilized **planutils** [10], while A* search with the goal count heuristic was executed using **downward** [11]. The table below shows the results obtained from these planners, with the best and worst performances highlighted in green and red, respectively.

Scenario	Planner	Search time (s)	Plan length
simple	*ff*	0.00	34
	lama	2.73	34
	lama-first	0.75	34
	goal-count	0.83	34
crane	*ff*	0.00	57
	lama	7.62	57
	lama-first	0.65	57
	goal-count	1.35	57

Overall, in both scenarios, the FF planner consistently achieved the fastest search time, while the LAMA planner had the longest search time among the options. The plan length remained the same for all planners in both scenarios.

4.2. Problem 2

In the case of the second problem in the "simple" scenario, we employed the FF, LAMA, and LAMA-FIRST planners while excluding A* due to its excessive computational cost.

Scenario	Planner	Search time (s)	Plan length
simple	*ff*	0.01	35
	lama	0.00	37
		0.00	33
		0.04	29
		1564.81	27
	lama-first	0.84	37

The LAMA planner has presented 4 distinct solutions in which the plan length cost progressively decreases, as evidenced in the table.
In general, the fourth solution provided by the LAMA planner seems to offer the optimal plan length. This solution utilizes the Lazy WA* algorithm with the "hff" and "hlm" heuristics, which also act as preferred heuristics for node expansion. The weight parameter is set to 2, indicating the heuristic estimate's relative importance in the search process. However, despite its effectiveness in generating optimal plan, this solution tends to have a longer search time, making it computationally expensive. Therefore, comparing the performance of different solutions in terms of both plan length and search time, the LAMA-FIRST solution performs the worst, and on the other hand, the second LAMA solution, which employs the same aforementioned algorithm but with a weight parameter set to 5, provides the best balance between plan length and search time.
In the "fluents" scenario, we performed experiments using different planners on a variety of problems. To handle the inclusion of numeric fluents, we selected **ENHSP-2020** (Expressive Numeric Heuristic Search Planner) [12] as our planner system due to its capability in handling this requirement. We conducted tests with various planners and documented the outcomes in the following table.

Scenario	Planner	Search time	Plan length	States Eval.
fluents	*sat-airb*	0.659	24	275
	sat-hadd	0.185	25	605
	sat-hmrp	0.301	25	188
	sat-hmrph	0.112	25	126
	sat-hmrphj	0.151	25	126
	sat-hradd	0.278	25	605
	opt-blind	94.518	23	3745175
	opt-hmax	150.657	23	3291367
	opt-hrmax	146.74	23	3291367

As we can see from the table, we have tested our experiments with the following planners[4]:

- **sat-aibr**: A* plus AIBR heuristic that is the main configuration for planning with autonomous processes;
- **sat-hmrp**: Greedy Best First Search plus MRP heuristic;
- **sat-hmrph**: same as before but with helpful actions;
- **sat-hmrphj**: same as before but with not only helpful actions, but also helpful transitions;
- **sat-hadd**: Greedy Best First Search with numeric hadd;
- **sat-hradd**: as the previous, but every pair of numeric conditions is augmented with the implied redundant constraint;
- **opt-blind**: baseline blind heuristic that gives 1 to state where the goal is not satisfied and 0 to state where the goal is satisfied;
- **opt-hmax**: A* with hmax numeric heuristic;
- **opt-hrmax**: same as before, but with redundant constraints.

In terms of overall performance, it is evident that the last three planners yield the optimal plan. However, when considering factors such as search time, plan length, and states evaluated, it becomes apparent that the *sat-aibr* planner outperforms the others, delivering the optimal performance. On the other hand, *opt-hmax* is the most resource-intensive and time-consuming among the available options.

4.3. Problem 3

For the third problem, we did not need to compare various planners since we utilized an open-source HTN planner called **PANDA**[5], developed by the University of ULM, to conduct experiments on two distinct problems. Our main focus was comparing different settings regarding the possibility of defining a hierarchy for executing the goals, as well as examining the planner's performance in terms of achieving optimal solutions.

Scenario	Planner	Order	Search time (s)	Plan length
htn1	*PANDA*	Yes	1.675	30
htn2	*PANDA*	No	105.628	30

During our experimentation, we observed that introducing a hierarchy for the goals significantly aided the planner in reaching the optimal solution more rapidly.

[4]To delve deeper, refer to ENHSP-2020 planners and how to use them.
[5]It can be retrieved from this website.

4.4. Problem 4

In the fourth problem, we employed **OPTIC** [5], a specialized temporal planner tailored for problems involving time-dependent goal collection costs. By utilizing OPTIC, we effectively tackled the unique challenges presented by our problem domain.

Scenario	Planner	Settings	Search time (s)	Plan cost (s)
simple	*OPTIC*	-N	0.18	71.026
		-N -W1,1	0.18	71.026
		-N -E -W1,1	411.77	59.024
fluents	*OPTIC*	-N	0.30	62.021
		-N -W1,1	0.26	62.021
		-N -E -W1,1	73.06	38.017

Our experiments consisted of two distinct scenarios, providing us with the opportunity to assess the planner's performance under different conditions. Furthermore, we delved deeper into the capabilities of OPTIC by exploring three distinct settings within the planner:

- **-N**: it avoids optimizing initial found solution and runs weighted A* with W = 1.000, not restarting with goal states;
- **-N -W1,1**: it runs weighted A* with W = 1.000, not restarting with goal states;
- **-N -E -W1,1**: it runs weighted A* with W = 1.000, not restarting with goal states, while skipping EHC: go straight to best-first search.

Through our exploration, we obtained valuable insights into the performance of OPTIC in these varied contexts. Interestingly, in both scenarios, we were able to achieve optimal solutions in terms of plan cost. However, it is worth noting that these optimal solutions turned out to be significantly more expensive compared to the suboptimal solutions. This outcome underscores the complexity and nuances of planning problems. Achieving optimality in one aspect, such as plan cost, does not guarantee overall efficiency or effectiveness.

4.5. Problem 5

The plans for Problem 5 have been generated using the **PlanSys2** planner [6], which operates within the ROS2 infrastructure. By default, PlanSys2 employs the Partial Order Planning Forwards (POPF) planner [13]. However, both the PlanSys2 planner and the Panda planner lack the capability to perform a comprehensive search in the solution space. Consequently, only a single plan can be returned, and the search process terminates at that point. The scenarios in this problem are based on the same domain PDDL files as Problem 4, so we anticipate a similar solution to be computed. Unfortunately, the obtained plan is sub-optimal and requires the same amount of time as the plan from Problem 4.

5. ZIP FOLDER CONTENT

The zip archive contains 5 main folders (**problem1**, **problem2**, **problem3**, **problem4**, **problem5**), a **README.md** file and two PDF files, **assignment** and this report.
The first folder (**problem1**) contains two different scenarios:

- **simple**
  - domain file: **domain1_simple.pddl**;
  - problem file: **problem1_simple.pddl**;
  - a folder **results** containing **4 log files**;
- **crane**
  - domain file: **domain1_crane.pddl**;
  - problem file: **problem1_crane.pddl**;
  - a folder **results** containing **4 log files**;

In **problem2**, we find the following subfolders:

- **simple**
  - domain file: **domain2_simple.pddl**;
  - problem file: **problem2_simple.pddl**;
  - a folder **results** containing 6 log files;
- **fluents**
  - domain file: **domain2_fluents.pddl**;
  - problem file: **problem2_fluents.pddl**;
  - a folder **results** containing 9 log files;

The third main folder is structured as follows:

- **htn1**
  - domain file: **domain3_htn1.hddl**;
  - problem file: **problem3_htn1.hddl**;
  - a folder **results** containing only **htn1.log** file;
- **htn2**
  - domain file: **domain3_htn2.hddl**;
  - problem file: **problem3_htn2.hddl**;
  - a folder **results** containing only **htn2.log** file;

Also in **problem4**, there are two subfolders listed below:

- **simple**
  - domain file: **domain4_simple.pddl**;
  - problem file: **problem4_simple.pddl**;
  - a folder **results** containing 3 log files;
- **fluents**

  – domain file: **domain4_fluents.pddl**;
  – problem file: **problem4_fluents.pddl**;
  – a folder **results** containing 3 log files;

The folder **problem5** is the one with the most different structure. Inside we find **plansys2_problem5_simple** and **plansys2_problem5_fluents**, for the "simple" and "fluents" scenario respectively, each containing the following files and subfolders:

- **CMakeLists.txt**, responsible for configuring the build process of our software project;
- **package.xml**, used to define package metadata and dependencies;
- **terminal1.sh**, containing the instructions to run the code in the first terminal;
- **terminal2.sh**, with the instructions to run the code in the second terminal;
- a **src** folder where we store the implementation files (*.cpp*) for our C++ code;
- **launch/plansys2_problem5_launch.py**, which contains the code to select the domain and run the executables that implement the PDDL actions;
- a **pddl** folder containing **problem5_domain.pddl**, as domain file, and **problem5_problem** which includes all the necessary instances, predicates and goals for our problem;
- a folder **results** containing one log file for both terminals (**terminal1.log** and **terminal2.log**).

6. CONCLUSIONS

The objective of our project was to explore the application of PDDL and different planning strategies in addressing an emergency services logistics problem. We successfully modeled and solved the problem using PDDL and various planners, analyzing the results obtained in different scenarios. In this report, we presented a comprehensive overview of the problem, its dimensions, and the major assumptions made. We discussed the design choices we implemented providing a detailed list of actions and predicates used in each task.
In conclusion, our project demonstrated the feasibility of using PDDL and planners to tackle emergency services logistics problems. The results obtained, coupled with the identified areas for improvement, provide a foundation for future research aimed at enhancing the coordination, efficiency, and performance of robotic agents in real-world scenarios. Overall, the project serves as a proof-of-concept for the use of automated planning techniques in solving complex real-world problems and establishes a foundation for future research and development.

7. REFERENCES

[1] Haslum P., et al., *An introduction to the planning domain definition language*. Synthesis Lectures on Artificial Intelligence and Machine Learning, 2019, 13.2: 1-187.

[2] Holler D., et al., *HDDL: An extension to PDDL for expressing hierarchical planning problems*. In: Proceedings of the AAAI Conference on Artificial Intelligence. 2020. p. 9883-9891.

[3] Fox M., Long D., *PDDL 2.1: An extension to PDDL for expressing temporal planning domains*. Journal of artificial intelligence research, 2003, 20: 61-124.

[4] Georgievski I., Aiello M., *An overview of hierarchical task network planning*. arXiv preprint arXiv:1403.7426, 2014.

[5] Benton J.; Coles Amanda, Coles Andrew, *Temporal planning with preferences and time-dependent continuous costs*. In: Proceedings of the International Conference on Automated Planning and Scheduling. 2012. p. 2-10.

[6] Martìn F., et al., *Plansys2: A planning system framework for ros2*. In: 2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, 2021. p. 9742-9749.

[7] Macenski S., et al., *Robot Operating System 2: Design, architecture, and uses in the wild*. Science Robotics, 2022, 7.66: eabm6074.

[8] Hoffmann J., *FF: The fast-forward planning system*. AI magazine, 2001, 22.3: 57-57.

[9] Richter S., Westphal M., *The LAMA planner: Guiding cost-based anytime planning with landmarks*. Journal of Artificial Intelligence Research, 2010, 39: 127-177.

[10] Muise C., et al., *PLANUTILS: Bringing Planning to the Masses*. In: 32nd International Conference on Automated Planning and Scheduling, System Demonstrations and Exhibits. 2022.

[11] Helmert M., *The fast downward planning system*. Journal of Artificial Intelligence Research, 2006, 26: 191-246.

[12] Scala E., et al., *Interval-based relaxation for general numeric planning*. In: ECAI 2016. IOS Press, 2016. p. 655-663.

[13] Coles A., et al., *Forward-chaining partial-order planning*. In: Proceedings of the International Conference on Automated Planning and Scheduling. 2010. p. 42-49.