APTP: AUTOMATED PLANNING THEORY AND PRACTICE
ACADEMIC YEAR 2022-2023

UNIVERSITY OF TRENTO - ARTIFICIAL INTELLIGENCE SYSTEMS

Alessandro Zinni - Pierluca Faccin
alessandro.zinni@studenti.unitn.it - pierluca.faccin@studenti.unitn.it

ABSTRACT

The aim of this project is to explore the challenges of automated planning in dynamic and resource-constrained environments, with a focus on real-world applications such as robotic agents. By encoding the problem classes in a powerful planning language, such as PDDL, we aim to demonstrate the ability of planners to effectively handle complex scenarios. The primary objective of this project is to establish a foundation for further research and development by incorporating multiple constraints and complexities.
To accomplish this, we have used PDDL and PlanSys2 as our primary tools, and have focused on addressing five key problems. These problems primarily focus on the delivery of essential goods such as food, medicine, and tools during emergency situations by an autonomous robotic agent in time constraint situations.
This project serves as a proof of concept for more extensive applications in the future.

1. INTRODUCTION

This report is an in-depth examination of the application of automated planning to the problem of emergency services logistics. Automated planning is a branch of Artificial Intelligence that uses a deliberative process to solve a problem by performing a sequence of actions.
In our problem, we consider the presence of autonomous robotic agents in decision-making by applying off-line planning, a procedure performed before actual execution that does not require particularly strict constraints on the computational or temporal resources needed to synthesize a plan. The problem we are addressing 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.
This report structure follows the same formalism proposed by the assigned problem. Considering given 5 problems, in **Section** 2 we will try to describe briefly the PDDL language and how it is structured, in **Section** 3 we will analyze the characteristics of the problems and our design choices, in **Section** 4 we will present our results, in **Section** 5 we will present the structure of our delivered zip-folder and, finally, we will conclude the reports with our thoughts and conclusion in **Section** 6.
All the code cited in here is present in the GitHub repository.

2. PDDL

PDDL (Planning Domain Definition Language) is a language used for defining planning tasks and domains. It is a standard language used for expressing the requirements and constraints of a problem in a format that can be understood by automated planning systems. PDDL is widely used in the field of artificial intelligence, particularly in the area of automated planning and scheduling.
The main components of PDDL include the domain file and the problem file. The domain file contains a description of the predicates, actions, and constraints of the problem. It also defines the types of objects that can be used in the problem and the relationships between the different objects. The problem file contains a description of the initial state and goal of the problem, as well as any additional constraints or requirements. Together, the domain file and problem file provide a complete description of the problem, which can be used by a planning system to generate a plan.
The requirements for PDDL, specified in the domain file, are known as the ”requirements of the planning system”. The most common requirements are the ”strips” and ”typing” requirements. The ”strips” requirement means that the planning system must be able to handle conjunctions and negations of predicates, while the ”typing” requirement means that the planning system must be able to handle different types of objects.
In Problem 3 we used **HDDL** (Hierarchical Task Network Domain Definition Language) instead of PDDL. HDDL is a

variation of PDDL that allows for the definition of hierarchies and methods, as well as the use of HTN (Hierarchical Task Network) in the problem file. HDDL is particularly useful for problems that require a hierarchical decomposition of tasks, such as multi-agent planning problems. The use of HDDL in this exercise was intended to demonstrate the ability of a planner to handle complex problems with a hierarchical structure.

3. PROBLEMS AND DESIGN CHOICES

This section of the report will present the various exercises that were conducted as part of this project. Each exercise addresses the problem of generating a plan in an automated fashion in real-world domains where time constraint actions and consumer resources must be taken into account. The modelled problem classes are based on real-world domains, in our case, robotic agents. As briefly mentioned earlier, to address these problems, we have used powerful planning languages such as PDDL and HDDL.
It is assumed that for all the exercises, the robotic agents always move with the carrier. Additionally, it is assumed that each box can be filled with one and only one item, as stated in the assignment text: "Each box is initially at a specific location and can be filled with a specific content such as food or medicine or tools, if empty." Furthermore, as stated in the assignment text, "move to another location moving the loaded box (if the box has been loaded, otherwise it simply moves);" we have handled the movement of the robot by creating two different move actions, one for when the robot is carrying a full box and one for when it is not. This allows for more control over the planning process and ensures that the robotic agent is able to deliver the appropriate boxes to the correct locations.

3.1. Problem 1

The problem at hand involves a robotic agent that needs to deliver boxes filled with essential goods such as food, medicine, and tools to specific locations where people are in need of them. The initial conditions of the problem state that all boxes and their contents are located at a single location known as the depot. Additionally, there are no injured people at the depot and the robotic agent is also located at the depot. The goal of the problem is to ensure that the people in need receive the contents they require by delivering the appropriate boxes to them and removing the contents from the boxes at the location. It is important to note that the goal is not to deliver specific content to specific people but rather to ensure that the people have access to the content they need. The robotic agent must take into account the lasting actions and consumer resources in order to accomplish this task.
We define the **depot** and the robotic **agent** as constants, while the types include **person**, **location**, **food**, **medicine**, **tool** and **box**. For each type, we define predicates that will be used in the actions such as *at-person*, *at-box*, *at-robot*, and *at-item* to locate the various types, *empty* and *full* to manage the state of the boxes, *loaded* and *free* to manage the state of the robotic agent, *need-food*, *need-medicine*, and *need-tool* to define the goal of the problem file later. Additionally, the predicate *inbox* serves to ensure that two boxes cannot be filled with the same item at the same time [1].
The actions in this problem include **fill-item**, **load-robot**, **move-with-box**, **unload-robot**, **empty-box-food**, **empty-box-medicine**, **empty-box-tool** and **move**. We modeled these actions in such a way that they pass the appropriate parameters, to ensure that the preconditions and effects create the desired planning. To verify this, we created in the problem file: 3 people, 2 locations, 3 boxes, 2 types of medicine, 1 type of food, and 1 type of tool.
It is important to note that, even though we use names such as medicine1, food1, tool1, for our project, it is not important that a person has food1, but rather that the person need-food. Therefore, the final goal of the problem is handled as (not (need-food person)).

3.2. Problem 2

The second problem of our project builds upon the previous problem, the Problem 1, by considering alternative methods of transportation for the boxes, such as using trucks, helicopters or drones. This means that the robotic agent can load up to four boxes onto a carrier, which must all be located at the same place. The robotic agent can then move the carrier to a location where people require supplies and unload one or more boxes from the carrier at that location. The robotic agent can continue moving the carrier to another location, unloading additional boxes and so on, without having to return to the depot until all boxes on the carrier have been delivered. Additionally, although a single carrier is needed for the single robotic agent, there is still a separate type of carrier.
To allow the carrier to move with more than one box we have introduced three more actions: **move-with-box2**, **move-with-box3**, **move-with-box4**. Which specify the movement of the robot with 2, 3 and 4 boxes respectively. These three actions have to be added to our solution because as an effect of **move-with-box** we update the position of the box to be at the new location. Therefore by having more than one box, we need to update the position of each box, requiring 4 different actions, one for each amount of boxes carried by the robotic agent.
For each robotic agent, we need to count and keep track of which boxes are on each carrier and how many boxes there are in total on each carrier, to prevent carriers from being overloaded.
To manage the number of boxes loaded on each carrier, we have introduced a new predicate *at-carrier* to track the car-

[1] Since the planning system has no knowledge about the semantics of the problem, every detail should be taken in consideration.

rier's position. Additionally, we have added the predicates **capacity** and **capacity-predecessor** to keep track of the number of boxes loaded on the robotic agent and prevent carriers from being overloaded.
We have differentiated two solutions for this problem: one that uses these two new predicates (i.e. capacity and capacity-predecessor) and is supported by fast-downward, and another solution that utilizes fluents (i.e. predicates with a numeric value) and increase and decrease effects for the fluents **carrier-load** and **carrier-capacity** which respectively indicate the number of boxes currently on the carrier and the total number of boxes that the carrier can hold. This solution is supported by an online planner. Furthermore, we have added **total-cost** that should allow the robotic agent to load more than one box at a time on the carrier, by reducing the cost as the number of boxes carried increases [2]. The only problem is that the planning system ignores silently the minimization objective specified in the problem file. It is worth mentioning that only with the version that makes use of fluents we were able to model the fact that the carrier can return to depot only if it has no boxes on it, because, without fluents, the capacity of the carrier is known only in the problem, therefore we are not able to model the fact that the carrier should be empty before returning to depot.
The first solution is specified in **domain-robot.pddl** and **pb-robot.pddl**, whereas the fluents solution is specified in **fluents-domain-robot.pddl** and **fluents-pb-robot.pddl**. A simpler version of the problem for the fluents part is specified in **fluents-simple-pb.pddl**
The initial conditions and goals remain unchanged from Problem 1, and we have simply adapted the actions by introducing the carrier and passing the appropriate parameters, preconditions, and effects for each action.

3.3. Problem 3

The third problem of our project builds upon the previous problem, the Problem 2, by addressing it using hierarchical task networks (HTNs). HTNs are a planning method that allows for the decomposition of complex tasks into a hierarchy of subtasks. Each subtask can be further decomposed into a set of simpler tasks. This hierarchical structure allows for a more efficient planning process by breaking down the problem into smaller and more manageable parts.
To address the problem using HTNs, we have introduced different :tasks and :methods. Tasks are high-level actions that are used to achieve a goal, while methods are a way of specifying a decomposition of these higher-level tasks. (sub)Tasks that cannot be further decomposed, are known as actions (same as actions used in problems 1 and 2).
In this case, we have defined tasks such as **deliver-food**, **deliver-medicine**, **deliver-tool**, **prepare-box**, and **give_box**, which are used to manage our goals and ensure the correct boxes with respective items are delivered. The methods we linked to these tasks are: *m_deliver_ordering_food*, *m_deliver_ordering_medicine*, *m_deliver_ordering_tool*, *m_prepare_box*, *m_give_box_food*, *m_give_box_medicine*, *m_give_box_tool*.
In the domain file, we have defined :htn, which is a way to define our problem's goal with a specific ordering. The :ordering parameter specifies the order in which the methods should be executed to achieve the goal. This allows for more control over the planning process and can lead to more efficient solutions. It's important to notice that the scenario remains the same as the previous problem, as well as the actions. The only difference is the way we address the problem using HTNs and the specific tasks and methods defined for this problem.
The **Fig** 1 illustrates a diagram of how we have decided to construct our hierarchical implementation of the actions. It provides a visual representation of how the tasks and methods are organized and how they interact with each other to achieve the desired goal.

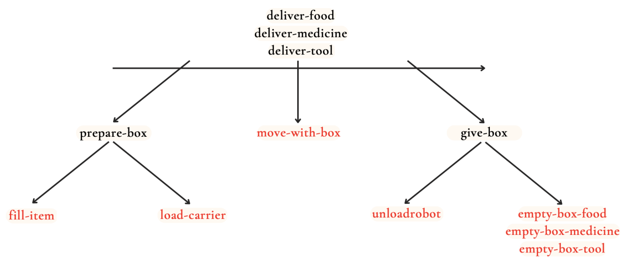

Fig. 1: Diagram actions: In black are written the tasks and in red are written the ground actions

It is important to notice that by using this decomposition, actions **move-with-box2**, **move-with-box3**, **move-with-box4** are really difficult to integrate. To this end, we have decided to stick with only one box that can be carried by the robotic agent. A simple solution may be given by removing the deliver method, leaving only **prepare-box** and **give-box** as higher-level tasks, and by specifying every single task in the problem (e.g. prepare-box1, prepare-box2, move-with-box2 ....), it would have been easier to account for the movement with multiple boxes, but in our opinion, this implementation would have ruined the hierarchical representation which we considered more important (in this problem), than the action of carrying more than one box.

3.4. Problem 4

The fourth problem of our project builds upon Problem 2, by introducing the use of durative actions. Durative actions are a type of action that have a duration, rather than being instantaneous (so duration should be specified when declaring a du-

[2] This is done only to see whether the agent was able to move with more than one box

rative action). There is a key difference between actions and durative actions which is the fact that they do not have preconditions anymore, but conditions. This is done because some of them may be true at the end of the action and also during the whole execution of the action. To address this change, three temporal expressions have been added to PDDL: overall, at start, and at end, which represent the state of the world during the overall duration of the action, what the state of the world is at the start of the action, and what the state of the world is at the end of the action, respectively. Temporal expressions have been added to effects as well. To account for the added complexity that emerges from the introduction of durative actions, we have introduced several new predicates such as **delivered**, **filling**, **loading**, **emptying**, **unloading**, and **equalbox**. Additionally, we have taken into account the possibility of having actions executed in parallel when it would be possible in reality, as this allows for a more realistic and efficient planning process. To do this, we have assigned arbitrary but reasonable durations to the different actions, such as *fill-item duration = 5*, *load-carrier duration = 4*, *move-with-box duration = 10*, *move-with-box2 duration = 10*, *unloadrobot duration = 2*, *empty-box-(food/medicine/tool) duration = 3*, in order to handle situations where multiple actions are happening at the same time, and to make sure that the planner doesn't create inconsistencies like filling two boxes at the same time with the same item. In this way, the parallel execution of actions is possible and it allows us to minimize the total execution time. An example of parallel actions in our solution: once a box is filled with a specific item, it can be loaded onto the robotic agent. Since we may have multiple boxes, the action of loading them onto the carrier can be done in parallel, meaning the loading of box 1 and box 2, with food and medicine respectively (for example), onto the same carrier, occurs simultaneously.

3.5. Problem 5

This report presents the solution we developed for Problem 5, which involved implementing Problem 4 with the PlanSys2 framework. **PlanSys2** is a powerful planning system that allows to model and solve complex planning problems, such as those involving multiple agents, resources, and constraints. Plansys2 is implemented in (**ROS2**). ROS2 is an open-source framework for developing robotics applications. It provides a set of libraries and tools for designing and implementing robot software, including libraries for communication, control, perception, and more. By using ROS2, we were able to take advantage of its powerful capabilities to integrate our planning system with other components of the robot, such as sensors, actuators, and controllers.
Our implementation started with a simple example and we made the necessary modifications to create the complete environment as described in detail in section 5.
In order to achieve this, we implemented a set of Application Programming Interfaces (APIs) in C++, such as:

- *fill_item_node.cpp*
- *load_carrier_node.cpp*
- *unloadrobot_node.cpp*
- *move_node.cpp*
- *move_with_box_node.cpp*
- *move_with_box2_node.cpp*
- *move_with_box3_node.cpp*
- *move_with_box4_node.cpp*
- *empty_box_food_node.cpp*
- *empty_box_medicine_node.cpp*
- *empty_box_tool_node.cpp*

to simulate the actions that the robot would perform. These actions were simulated by calling the API and simulating the action for a predefined duration of n milliseconds.
As we did not have access to a physical or virtual robot for testing, these actions are referred to as ”fake actions” in our system. These APIs were developed to serve as a testing pipeline for API calls and their corresponding responses.

4. RESULTS

For the sake of simplicity we decided to include the results inside one file in each problem folder, containing the output of the planner. It is important to notice that results of problem 2 and problem 4 are obtained by running a simpler version of the problem file (Can be found under the name *pb-simple.pddl*) due to the fact that the full problem is really heavy to run. Nevertheless it should work even with the full version (i.e. *pb-robot.pddl*). Results are inside the file *results.txt* in every folder of the corresponding problem. If version with fluents is provided, the results file is called *results-fluents.txt*.

5. ZIP FOLDER CONTENT

The zip archive contains 5 different main folders (named **Problem1**, **Problem2**, **Problem3**, **Problem4**, **Problem5**) and the README.md file. The first folder contains the files [*domain-robot.pddl*, *pb-robot.pddl*] and the corresponding output file. The second folder contains 2 versions: [*domain-robot.pddl* and *pb-robot.pddl*] which is the non-fluent version and [*fluents-domain-robot.pddl* and *fluents-pb-robot.pddl*] which is the fluent version with two files containing the corresponding output for each version. In addition there is a README.md that explains how to run the version with fluents. It is important to notice that there is also a simple version of the problem (see files *fluents-simple-pb.pddl* and *pb-simple-robot.pddl*) due to the fact that actions where the movement with more than one box have a lot of preconditions, the plan is really heavy to run, hence it is possible to run the simpler version to check that all is working fine. The third folder contains the files [*domain-robot.hddl* and *pb-robot.hddl*]. The fourth folder contains the files [*domain-robot.pddl*, *pb-robot.pddl* and *pb-simple.pddl*], again a simpler version is provided to facilitate the execution. The fifth folder is the one with the most different structure. Inside we find the simple example from which we started solving the solution:

- CMakeLists.txt
- package.xml
- launch/commands, which contains all the necessary instances, predicates and goals for our problem
- launch/launcher.py, which contains the code to select the domain and run the executables that implement the PDDL actions
- pddl/domain-simple.pddl, which contains the domain file of the Problem 4 in 3.4
- src folder, which contains all the C++ APIs implementation listed in Section 3.5

6. CONCLUSIONS

In conclusion, our project aimed to demonstrate the feasibility of planning actions for a robot operating within a facility, such as a hospital. We simulated a scenario in which the robot is responsible for delivering food and medicine to injured patients, starting from a storage area. By analyzing the problem and developing a solution, we were able to conclude that this problem can be extended and optimized in a variety of ways. One potential future scenario could involve a more sophisticated implementation of hierarchical task networks, allowing to specify a higher level **deliver** task that takes into account also actions for moving with more than one box at a time. An optimization can be introduced for example by using axioms to specify some recurrent patterns that are always true, this would allow the reduction of the size of the precondition part of some actions [3].
In summary, our project opens up a wide range of possibilities for the application of robotic systems in real-world scenarios, and we believe that further research in this area has the potential to yield significant improvements in efficiency and performance.

[3] Unfortunately axioms were not supported by fast downward.