AI for Robotics II
Automated Warehouse Modeling

Salterini Filippo, Amato Francesca, Polese Carolina
Robotic Engineering - University of Genoa

May 23, 2025

Abstract

This report presents a possible model of an automated warehouse solution as part of the *AI for Robotics II* course. The warehouse involves two mobile mover robots, whose purpose is to transport crates from their initial locations to a loading bay, and at least one fixed loader robot, responsible for putting them on a conveyor belt. The core objective was to model this system using efficient AI planning techniques studied during the course, trying to shape our model as realistically as possible, using constraints like weight, distances, and time-dependent actions

Additionally, the model incorporates several optional extensions to increase its realism and complexity. Our group implemented a model where all of them were included. In particular, we have implemented:

- **Groups handling**: some crates must be delivered together subsequently on the conveyor belt. A crate can belong to a specific group (eg, A and B) or no group.
- **Battery limits**: Each robot has a limited energy available to carry out the actions. The capacity is 20 power units, and decreases for each unit of time during which the robots are actively doing something. A charging bay is located at the base station where the robot can charge; the charging time is based on how much the robot has consumed the battery.
- **Fragile**: Some crates are fragile and need extra care to be moved around. This kind of crate needs both robots to carry it, and the loaders take more time to load it onto the belt.
- **Two loaders**: A second loader can help the first one. It uses the same loading bay and can work while the other one is occupied. But it can only load light crates.

Chapter 1

Introduction

Warehouse automation is a key application of artificial intelligence and robotics, aimed at improving efficiency, reducing costs, and enhancing logistics reliability. In the *AI for Robotics II* course, we were tasked with designing an automated warehouse system where robots collaborate to transport and load crates for delivery. The goal was to create an efficient and realistic model, suitable for AI planning.

The scenario involves physical constraints, robot coordination, and timing requirements. While two mobile robots move crates and a fixed loader places them on the conveyor belt, optional extensions introduce additional challenges, such as managing crate groups, handling fragile items, adding a second loader, and modeling robot energy consumption and recharging.

To address the problem, we used the OPTIC planner, a temporal planner that handles durative actions and numerical fluents, enabling us to manage timing constraints and optimize the plan duration. The aim was to capture a real-world system's limitations while ensuring solvability through modern AI planning tools. This report details the modeling process and key design choices to balance expressiveness and computational efficiency.

Chapter 2

Material and methods

To model our warehouse scenario, PDDL 2.1 was used. PDDL is a human-readable format that attempts to standardize Artificial Intelligence planning languages. This language isolates the domain, a structure that defines the universal aspect of the planning problem, like actions, predicates, and types, from the problem, in which specific instances of the domain are written, detailing the initial state and the desired goal state. PDDL 2.1 introduces the numeric fluents, plan metric, and durative actions, allowing for the representation of a more realistic model of the world and optimizing the plan generated.

2.1 PDDL - Domain

The `domain.pddl` defines the universal warehouse environment we came up with. Using this file, the solver, if well defined, can generate possible plans, if they exist, for each problem.

The `domain.pddl` specifies:

- **Objects**: such as robots, crates, bases, and loaders;
- **Predicates**: conditions that describe the properties of objects and their relationships.
- **Actions**: the possible actions that can be performed in the environment, considering if their preconditions are met and how their effects modify the status of the world.
- **Functions**: Numerical values that can be modified by the agents (battery, priority) or used as constants (distance between crate and loading bay).

In the following, the characteristics of our domain are explained.

2.1.1 Functions Used

In the `domain.pddl` file, the following functions are defined:

- **Distance**: Represents the distance between a crate and a base. This is critical for calculating movement durations.

- **Weight**: Indicates the weight of a crate, which influences whether a single robot can carry it or not and reduces the velocity of the robot when the crate is carried by the robots.
- **Velocity**: A constant function that represents the velocity of the robots and affects how quickly they can move (different values of velocities can be set in the `problem.pddl` to see how the plans would change).
- **Battery**: Tracks the battery level of a robot, which influences its ability to perform tasks, and it needs to be checked before a new cycle of actions to choose if it has to be charged or not.
- **Max Battery**: Maximum capacity of the robots battery.
- **Free Base**: Monitors the status of the base, if there's a crate in the loading bay, the value is increased by one, if a loader picks a crate the value is decreased by one. Robots can only deploy crates when the value is less than 2.
- **Priority**: Tracks the loading status of crates based on the group they are assigned to. First, process those without a group (*priority* = *max priority*), then those in Group A ($0 < priority < max\ priority$), and finally, those in Group B ($priority \leq 0$).
- **Max Priority**: Maximum value of priority defined in the problem.

These functions enable the planner to consider numeric constraints, such as movement times based on distance and battery consumption during actions.

2.1.2 Predicates

Here's a short description of the predicates instantiated in the domain:

Predicate	**Description**	**Extension**
`at_crate(r,c)`	Robot r is in the same position as crate c	Base
`at_base(r,b)`	The robot r is at base b	Base
`carry(r,c)`	The robot r is carrying crate c	Base
`free(r)`	The robot is free, performing no actions	Base
`at(c,b)`	The crate c is at base b	Base
`pickable_crate(c)`	The crate c is still in its initial position, not yet picked up by a robot	Base
`free_charger(b)`	The charging station at base b is available	Charging
`free_loader(l)`	The loader l is idle, performing no actions	Base
`free_loader_leggero(ll)`	The second loader ll is idle, performing no actions	Second Loader

Predicate	Description	Extension
`pick_load(l,c)`	The crate c has been picked up by loader l	Base
`pick_load_leggero(ll,c)`	The crate c has been picked up by the second loader ll	Second Loader
`on_belt(c)`	The crate c has been loaded onto the conveyor belt	Base
`fragile(c)`	The crate c is fragile	Fragile Crate
`not_fragile(c)`	The crate c is not fragile	Fragile Crate
`groupA(c)`	The crate c belongs to group A	Groups
`groupB(c)`	The crate c belongs to group B	Groups
`no_group(c)`	The crate c does not belong to any group	Groups
`not_working_together`	The two robots are not working together	Base
`working_together`	The two robots are working independently	Base

2.1.3 Actions

The `domain.pddl` defines multiple actions to manage warehouse operations. These include:

- ***Charging Cycle***: `charging` manage robot battery recharging. A robot can recharge when docked at a base, free and the charging base is not occupied by another robot. The robot releases the charging spot when it's fully charged, and its duration depends on the level of battery level of the robot.
- ***Single-Robot Handling***: `move_to_crate`, `pick_up`, `move_back`: handle crates transportable by a single robot. A cycle of actions can be started only when the battery level is enough, the crate can be picked, its weight is under 50 kg, and it's not fragile.
- ***Cooperative Handling for Heavy Crates***: `move_two_robot_to_crate`, `pick_two_robot`, `move_back_two_robot`, `move_back_two_robot_leggero`: enable two robots to cooperate on crates that exceed individual weight limits or when the crates are fragile. It has to be noted that a robot can go to a crate's location even if it is not fragile or weighs more than 50 kg; in that case, to go back, the pair of robots can choose to execute the next action listed here.
- ***Optimized Cooperative Handling for Light Crates***: `move_back_two_robot_leggero` is a faster and less energy-consuming version, used for light crates carried in pairs back to base.
- ***Standard Loading Operations***: `grasp`, `loading`: the main loader performs crate loading from base to conveyor.

- ***Fragile Loading Operation***: `loading_fragile`, the main loader performs fragile crate loading from base to conveyor; this action takes 6 units of time instead of 4, since the loader has to be more careful.
- ***Alternative Loading Path***: `grasp_leggero`, `loading_leggero`, `loading_leggero_fragile`: fallback sequence for crate loading using a secondary loader that can only load light crates (for non fragile crates it takes 4 units, while 6 units for fragile crates).
- ***Actions for Group Delivery***: Some crates belong to a group (A or B) and must be loaded onto the conveyor belt in sequence to assist the delivery person. This constraint is specified in the problem file. The following actions handle this constraint:
  - `drop_A`, `drop_B`: used when a single robot delivers a crate belonging to group A or B. `drop_B` can't be executed if `drop_A` has not decreased the value of priority till zero.
  - `drop_two_robot_A`, `drop_two_robot_B`: cooperative versions for delivering crates of group A or B with two robots.
  - `drop_two_robot` and `drop` are executed only when priority has the same value as max priority; therefore, a crate that does not belong to any group is deployed first.

2.2 PDDL - Problems

The `problem.pddl` describes a specific instance of the planning scenario: it specifies the objects involved, their initial state, and the goal to be achieved. Each problem file, therefore encodes a concrete challenge that the planner must solve using the actions defined in the domain.

In this project, five different problem instances have been designed, each increasing in complexity and intended to test different aspects of the planning domain. All problems assume that the movers begin at the loading bay and that the distances are straight-line measurements from the loading bay to each crate. There is no risk of collision or interference along the paths.

The table 2.2 provides a brief description of each problem instance.

The goal of each problem is that the existing crates have been processed by the loader, thus for each crate:

```
on_belt(crate)
```

Problem	Crate Type	Weight (kg)	Distance	Group
0.5	Regular	70	10	-
	Regular	20	20	A
1	Regular	70	10	-
	Fragile	20	20	A
	Regular	20	20	A
2	Regular	70	10	A
	Fragile	80	20	A
	Regular	20	20	B
	Regular	30	10	B
3	Regular	70	20	A
	Fragile	80	20	A
	Regular	60	30	A
	Regular	30	10	-
4	Regular	30	20	A
	Fragile	20	20	A
	Fragile	30	10	B
	Fragile	20	20	B
	Fragile	30	30	B
	Regular	20	10	-

Table 2.2: Summary of crate specifications for each problem instance.

Chapter 3

Solving and Analysis

Now that we have formulated our domain and the problems to solve, we can analyze the solutions and plans we have found. It follows a short description of the planning engine our group has chosen to accomplish this task, providing its pros and cons.

3.1 Planning Engine - OPTIC

Our group considered OPTIC the most suitable solver given the need to set the duration of actions, which depends on both the *distance* and the *weight* of the crates. OPTIC is a temporal planner designed to solve problems where the plan's cost depends on preferences or time-varying costs. Thus, we were able to use **durative-actions** within our domain. Moreover, OPTIC:

- Can handle complex constraints, such as time limits and soft preferences in planning.
- Uses heuristic algorithms to improve the efficiency of plan searches.

However, we also encountered some limitations of OPTIC, including:

- No PDDL+ support: OPTIC does not handle non-linear effects or discrete events occurring over time, which means we could not use events or processes that would have been useful for battery management.
- No negative preconditions support: This led to an increase in variables in our domain (e.g., `fragile` and `not_fragile`).
- No conditional effects support: Conditional effects would have provided greater flexibility in modeling complex problems, such as handling grouped crates or fragile crates.

Additional references to OPTIC can be found at the following page:

`https://planning.wiki/ref/planners/optic`

3.2 Analysis Of The Performance Of The Planning

The domain was tested using the OPTIC planner. The plans corresponding to different problems will be compared in order to highlight the planner's performance in various scenarios.

3.2.1 Statistic Analysis

Four key metrics will be analyzed to evaluate the quality and efficiency of the planner:

- **Makespan**: the total time elapsed from the start of the first action to the end of the last one.
- **States evaluated**: represents the number of distinct states the planner has evaluated to find a valid plan. A higher number means more computational effort, which may be caused by a higher complexity of the problem passed to the solver.
- **Planning time**: the actual computation time the planner required to explore the state space, apply heuristics, and return one or more solutions.
- **Ground Size**: all possible combinations of the parameters of each action with the constants. This section will be discussed separately in paragraph 3.2.2.

Problem 0.5

During the execution of `problem 0.5`, the planner found two temporal plans:

- Plan 1: 30.005 s
- Plan 2: 30.005 s

Parameter	Value	Description
Makespan	30.005	The total duration of the plan, from the start of the first action to the end of the last one.
States evaluated	35	The low number of evaluated reflects the fact that the problem 0.5 is quite simple.
Planning time	10.113 s	10 seconds is a good value for computing a plan.

Table 3.1: Performance metrics and explanations for problem 0.5

These results suggest that OPTIC repeatedly found the same optimal solution, highlighting its stability concerning this problem. The plan starts with the action of moving both robots toward crate1 (heavy, 70 kg) using: (`move_two_robot_to_crate robot2 robot1 crate1 base1`).
After the drop of crate1 at the base, the loading process onto the conveyor belt begins. Next,

robot1 moves alone to retrieve and transport crate2 (light, 20 kg), performs a drop at the base, and starts the loading process.

After the first drop, both robot1 and robot2 have 12 battery units remaining. Therefore, they need to visit the charging bay to recharge and then resume activity. This step is necessary to accumulate enough energy to perform the upcoming actions: `move_to_crate` and `move_back`.

The same applies to robot2: although it is not used in the following actions, it still needs to recharge at the charging bay.

Problem 1

The planner found only one plan:

- Plan1 = 44.673

Parameter	Value	Description
Makespan	44.673	Higher duration compared to the previous problem. However, crate handling is more complex, and one crate is fragile, requiring the use of a slower loading action (from 4 to 6 time units). The increased makespan is thus justified.
States evaluated	60	The number of evaluated states almost doubled compared to the previous problem, mainly due to the increased complexity and number of possible action sequences. Still an optimal value.
Planning time	32.485 s	Significantly increased planning time, proportional to the number of states and problem complexity.

Table 3.2: Performance metrics and explanations for problem1

Problem 2

The planner found only one plan, even if the solver evaluated, as can be seen in Table 3.3, 98 states.

Problem 3

The planner found three plans:

- Plan1 = 156.012
- Plan2 = 154.012
- Plan3 = 146.012

Parameter	Value	Description
Makespan	82.008	This is expected due to the presence of 4 crates (two of them heavy), requiring more serial actions.
States evaluated	98	The number increases moderately, indicating a well-structured domain that effectively constrains the search space.
Planning time	35.189 s	Despite the increase in states, the planning time remains stable. The added complexity does not significantly affect computation.

Table 3.3: Performance metrics and explanations for problem2 using OPTIC

Parameter	Value	Description
Makespan	156.012	Same number of crates as problem2, but the makespan is doubled due to the presence of very heavy crates requiring slow, time-consuming transport actions, and also fragile crates.
States evaluated	1376	A significantly higher number of paths were explored due to the increased complexity from varying action durations, energy requirements, and heavy crate handling.
Planning time	32.912 s	Despite the complexity, three valid plans were found within an optimal planning time.

Table 3.4: Performance metrics and explanations for problem3

However, all three plans share a common flaw: when processing crate3, a battery issue occurs. With a fully charged battery (20 units), it is not possible to complete the entire crate3 handling sequence.

Specifically:

- `move_two_robot_to_crate robot2 robot1 crate3 base1` lasts 3 units
- `move_back_two_robot crate3 base1 robot2 robot1` lasts 18 units

Total: 21 units, exceeding the battery capacity of 20.

To solve this issue, a possible solution is to increase the battery capacity maximum from 20 to 25 units, enabling the crate3 process to complete without violating energy constraints. Otherwise, the solver can't find a suitable plan.

Problem 4

Plans found by the planner:

- Plan1 = 101.681
- Plan2 = 89.681

Parameter	Value	Description
Makespan	101.681	Despite the significant complexity of the problem, we obtain a very good value, consistent with previous problems.
States evaluated	2449	A very high value compared to previous problems, due to the increase in combinations of available actions and resources. This highlights a moderate problem complexity, which leads to an increase in the time and memory required for planning, but also a higher efficiency in the executable plan.
Planning time	36.748 s	An acceptable value proportional to the complexity of the problem, considering the initial state, the goals, and the increase in resources compared to previous cases.

Table 3.5: Performance metrics and explanations for problem 4

These values indicate that the domain that was implemented is stable and efficient enough, thus the solver can produce at least a plan for each of the problems proposed in an almost constant planning time.

3.2.2 Ground Size

The ground size (or grounded size) of a PDDL domain refers to the total number of instantiated actions that the planner must consider for all the possible combinations of the parameters of each action with the objects available in the problem.

It depends on the following factors:

- The number of actions defined in the domain.
- The number and types of parameters for each action.
- The number of objects in the problem corresponding to those types.

Assuming:

$$n_r = number\ of\ robots$$

$$n_c = number\ of\ crates$$

$$n_b = number\ of\ bases$$

$$n_l = \textit{number of loaders}$$

$$n_{ll} = \textit{number of cheap loaders}$$

In our case, we can also assume that:

$$n_b = n_l = n_{ll} = 1$$

But the number of bases and loaders can be increased since it depends on the number of objects instantiated in the problem.

The total ground size computed for each problem is reported in Table 3.7.
General formulas for each action are leasted in the following table:

Action name	**Grounding formula**
`charging`	$n_r \times n_b$
`move_to_crate`	$n_r \times n_c \times n_b$
`move_back`	$n_r \times n_c \times n_b$
`pick_up`	$n_r \times n_c$
`drop`	$n_r \times n_c \times n_b$
`drop_A`	$n_r \times n_c \times n_b$
`drop_B`	$n_r \times n_c \times n_b$
`move_two_robot_to_crate`	$n_r \times (n_r - 1) \times n_c \times n_b$
`move_back_two_robot`	$n_r \times (n_r - 1) \times n_c \times n_b$
`move_back_two_robot_leggero`	$n_r \times (n_r - 1) \times n_c \times n_b$
`pick_up_two_robot`	$n_r \times (n_r - 1) \times n_c$
`drop_two_robot`	$n_r \times (n_r - 1) \times n_c \times n_b$
`drop_two_robot_A`	$n_r \times (n_r - 1) \times n_c \times n_b$
`drop_two_robot_B`	$n_r \times (n_r - 1) \times n_c \times n_b$
`grasp`	$n_c \times n_b \times n_l$
`loading`	$n_c \times n_b \times n_l$
`loading_fragile`	$n_c \times n_b \times n_l$
`grasp_leggero`	$n_c \times n_{ll} \times n_b$
`loading_leggero`	$n_c \times n_{ll} \times n_b$
`loading_leggero_fragile`	$n_c \times n_{ll} \times n_b$

Table 3.6: Ground size formulas for each individual action

We can notice that the formulas to calculate the ground size of the actions involving the two robots' movement in pairs are a bit different from the others. One could assume that the formula would have been like this:

$$n_r \times n_r \times \dots$$

But in this way we would include in the calculation also the case where $r_1 = r_2$, meaning that a single robot can accomplish the action of moving, picking, and dropping that are meant to be done by two robots. Since we want two different robots to execute these kinds of actions, the formalas has this structure such that:

- The first robot r_1 can be chosen in n_r ways.
- The second robot is chosen between the remaining $n_r - 1$

Problem	**Total Ground Size**
Problem 0.5	66
Problem 1	99
Problem 2	132
Problem 3	132
Problem 4	198

Table 3.7: Total ground size computed for each problem instance

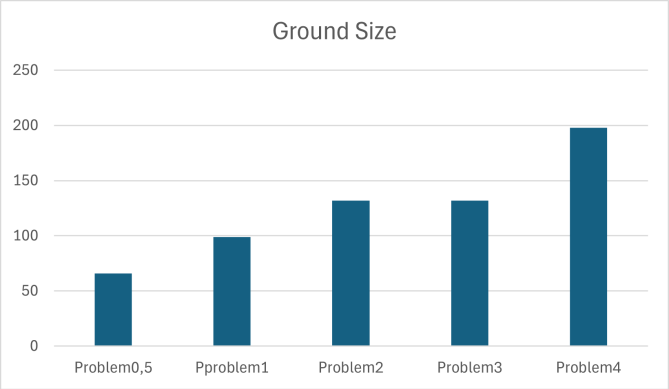

Table 3.8: Graph showing increasing ground size

As expected, an increase in the objects `crate` leads to an increase in the ground size. It can be noticed that since $n_r, n_b, n_l$and n_{ll} are constant values in our problem, the ground size is solely dependent on the number of the object `crate` instantiated in the problem and it's proportional to it. Thus, we can conclude that for any problem that has these characteristics, the ground size is given by:

$$ground\ size = 33 \times n_c$$

Chapter 4

Heuristics and Patterns

4.1 Heuristic Design

Once the domain and problem formulations are validated and solvable through an automated planner, the next step is to explore heuristic strategies that can guide the search more efficiently. This section presents a custom heuristic tailored to the simplified instance Problem 0.5 and discusses its expected behavior, admissibility, and usefulness. Additionally, a symbolic pattern abstraction is proposed to further analyze the solution structure. We define a state evaluation heuristic function $h : S \rightarrow N$, which estimates the cost of reaching the goal of a given state S. The goal is to load all crates onto the conveyor belt. In Problem 0.5, there are two crates with different weights and distances from the loading bay.

4.1.1 Description of the heuristic function

The heuristic is designed to estimate the cost to reach the goal by evaluating, for each crate not yet loaded:

- The cost for the robot(s) to pick and move the crate, depending on its weight and distance. This handling cost includes the estimated time to reach, lift, carry back, and deposit the crate.
- The recharging cost, proportional to the battery charge needed if the robot requires recharging before proceeding;
- Priority bonuses, where crates that do not belongs to any group receive a negative bonus to be loaded first. The same is done with the crates belonging to group A, such that $cost(no\_group) \leqslant cost(groupA) \leqslant cost(groupB)$
- The loading cost, which depends on the crate's fragility, set to 6 for fragile crates and 4 for normal ones.

The final value of the heuristic is the sum of all these costs for each crate still to be managed in the current state.

4.1.2 Motivation and property

Heuristics are useful because they guide the search towards states that:

- Optimize the energy management of robots;
- Prioritize the packages that need to be delivered first;
- Reduce loading costs by choosing between alternative strategies.

Heuristic Cost Evaluation

Initial conditions:

`distance_crate1` = 10
`distance_crate2` = 20
`weight_crate1` = 70
`weight_crate2` = 20
`velocity` = 10
`not_fragile_crate1, not_fragile_crate2` = `True`
`pickable_crate1, pickable_crate2` = `True`
`group_A_crate1` = `False`
`group_A_crate2` = `True`
`battery_robot1, battery_robot2` = 20
`at_base_robot1, at_base_robot2` = `True`
`free_robot1, free_robot2, free_base, free_charger, free_loader` = `True`

Goal:

`on_belt_crate1, on_belt_crate2` = `True`

$h = 0$

for all crate c not yet on the belt **do**
  if battery(robot) is low **then**
    $charge\_cost = max\_battery - battery(robot)$
  else
    $charge\_cost = 0$
  end if
  if c does not belong to any group **then**
    $group\_cost = -5$
  else if c belongs to group A **then**
    $group\_cost = 0$
  else
    $group\_cost = 5$
  end if
  if c is heavy OR fragile **then**
    $move$ = distance$(robot, c)$ / velocity
    $pick = 1$
    $move\_back$ = (distance$(robot, c) \times$ weight(c)) / 100
    $drop = 1$

$cost\_mover = move + pick + move\_back + drop$
if c is fragile **then**
$cost\_loader = 6$
else
$cost\_loader = 4$
end if
else if c is light **then**
$move = \text{distance}(robot, c) \,/\, \text{velocity}$
$pick = 1$
$move\_back = \text{distance}(robot, c) \times \text{weight}(c) \,/\, 100$
$drop = 1$
$time\_one\_robot = move + pick + move\_back + drop$
$move\_two\_robot = \text{distance}(robot, c) \,/\, \text{velocity}$
$move\_back\_two\_robot = \text{distance}(robot, c) \times \text{weight}(c) \,/\, 150$
$time\_two\_robot = move\_two\_robot + pick + move\_back\_two\_robot + drop$
$cost\_mover = \min(time\_one\_robot, time\_two\_robot)$
$cost\_loader = 4$
end if
$h = h + cost\_mover + cost\_loader + charge\_cost + group\_cost$
$battery(robot) - = cost\_mover$
end for
return h

This algorithm returns us the value of the heuristic (including as much as possible the extensions granted), for problem 0.5, the corresponding value is:

$$Crate1$$

$$\text{cost\_mover} = \frac{10}{10} + 1 + \frac{10 \times 70}{100} + 1 = 1 + 1 + 7 + 1 = 10$$

$$\text{cost\_loader} = 4$$

$$\text{group\_cost} = -5$$

$$h_1 = 10 + 4 - 5 = 9$$

Assuming that the battery has decreased after the loading of the crate1 by 10 units, now we have to consider the cost of charging:

$$Crate2 \textit{ (two possible values)}$$

$$\text{charge\_cost} = 20 - 10$$

$$\text{cost\_mover} = \frac{20}{10} + 1 + \frac{20 \times 20}{100} + 1 = 2 + 1 + 4 + 1 = 8$$

$$\text{cost\_loader} = 4$$

$$\text{group\_cost} = 0$$

$$h_{21} = 10 + 8 + 4 = 22$$

$$\text{charge\_cost} = 2 \times (20 - 10) = 20$$

$$\text{cost\_mover} = \frac{20}{10} + 1 + \frac{20 \times 20}{150} + 1 = 2 + 1 + \frac{40}{15} + 1 \approx 6.67$$

$$\text{cost\_loader} = 4$$

$$\text{group\_cost} = 0$$

$$h_{22} = 20 + 6.67 + 4 \approx 31$$

Since our algorithm selects the minimum value between h_{21} and h_{22}, h_{21} is the one considered for the second crate. That means that the cost for loading the second crate using just one robot is less than using two robots that move faster when carrying light crates. Final value of heuristic:

$$h = h_1 + h_{12} = 9 + 22 = 31$$

4.1.3 Relaxed Version

In addition to the pseudocode defining our main heuristic, we have also developed a relaxed version of this evaluation function. The relaxed heuristic comes from simplifying the original domain by ignoring some conditions or constraints to estimate the cost more quickly. For example, it might be assumed that the robot does not need to recharge or that the movement cost is calculated without weight considerations or fragile and group conditions over crates. This allows for a faster heuristic estimate that remains admissible, meaning it never overestimates the true cost.

This relaxed heuristic can serve as a lighter benchmark to guide the search, balancing between accuracy and calculation speed.

Algorithm 1 Heuristic Cost Evaluation (Relaxed Version)

Initial conditions:
    `distance_crate1` = 10
    `distance_crate2` = 20
    `pickable_crate1, pickable_crate2` = `True`
    `at_base_robot1, at_base_robot2` = `True`
    `free_robot1, free_robot2` = `True`
    `free_base, free_loader` = `True`
Goal:
    `on_belt_crate1, on_belt_crate2` = `True`
$h = 0$
for all crate c not yet on the belt **do**
    $move$ = distance(robot, c) $/\ velocity$
    $pick = 1$
    $move\_back$ = distance(robot, c) $/\ velocity$
    $drop = 1$
    $cost\_mover = move + pick + move\_back + drop$
    $cost\_loader = 4$
    $h = h + cost\_mover + cost\_loader$
end for
return h

The resultant heuristic for each crate is:

$$Crate1$$

$$\text{cost\_mover} = \frac{10}{10} + 1 + \frac{10}{10} + 1 = 1 + 1 + 1 + 1 = 4$$

$$\text{cost\_loader} = 4$$

$$h_1^* = 4 + 4 = 8$$

$$Crate2$$

$$\text{cost\_mover} = \frac{20}{10} + 1 + \frac{20}{10} + 1 = 2 + 1 + 2 + 1 = 6$$

$$\text{cost\_loader} = 4$$

$$h_2^* = 6 + 4 = 10$$

Thus,

$$h^* = h_1^* + h_2^* = 8 + 10 = 18$$

This relaxed version of the heuristic is admissible since its cost is less than the actual cost, such that:

$$h* \leq h$$

4.2 Pattern section

A pattern in planning is a symbolic sequence of actions or a subset of actions that represents a relevant part of the overall plan to solve a specific problem. Patterns are used to simplify problem analysis and solving by limiting the search space to meaningful and often repetitive action sequences. A good pattern should be simple yet representative enough to cover the key steps needed to reach the goal, thus helping to reduce computational complexity.
In the specific case of problem 0.5, the pattern we defined describes a series of coordinated actions between two robots and the loaders to handle the two crates.

```
Pattern:

1. move_two_robot_to_crate
2. pickup_two_robot
3. move_back_two_robot
4. drop_two_robot

5. grasp
6. loading

7. charging

8. move_to_crate
9. pickup
10. move_back
11. drop_A

12. grasp_leggero
13. loading_leggero
```

Applied to problem 0.5 :

```
Pattern:

1. move_two_robot_to_crate(robot1, robot2, crate1)
2. pickup_two_robot(robot1, robot2, crate1)
3. move_back_two_robot(robot1, robot2, crate1)
```

```
4. drop_two_robot(robot1, robot2, crate1)

5. grasp(loader, crate1)
6. loading(loader, crate1)

7. charging(robot1)

8. move_to_crate(robot1, crate2)
9. pickup(robot1, crate2)
10. move_back(robot1, crate2)
11. drop_A(robot1, crate2)

12. grasp_leggero(loader_leggero, crate2)
13. loading_leggero(loader_leggero, crate2)
```

As we can see, the bound for this pattern is equal to 1.
We also create a pattern for the relaxed version of our domain, studying the states and the actions that can be executed in the current state:

```
Pattern relaxed heuristic:

1. move_to_crate
2. pickup
3. move_back
4. drop
5. grasp
6. loading
```

```
Pattern relaxed domain:

1. move_to_crate(robot1, crate1)
2. pickup(robot1, crate1)
3. move_back(robot1, crate1)
4. drop(robot1, crate1)
5. grasp (loader, crate1)
6. loading(loader, crate1)

7. move_to_crate(robot2, crate2)
8. pickup(robot2, crate2)
9. move_back(robot2, crate2)
10. drop(robot2, crate2)
11. grasp (loader, crate1)
12. loading(loader, crate1)
```

In this case, since we have removed some restrictions and simplified our domain, the movers have a limited number of simple actions they can do; therefore, the bound has increased to 2.

Chapter 5

Results and Conclusion

We find this project quite interesting to develop, showing us how a proper model of the world could help to solve complex logistics tasks in a warehouse environment, incorporating AI. Our projects, furthermore, can generate one or more plans in an almost stable period of 30 seconds, enabling the possibility of running in real time each time a new order is received.

The heuristic evaluation and pattern analysis further enriched the planning process, offering insight into the trade-offs between computational efficiency and model accuracy. These tools proved to be effective in guiding the planner and understanding the behavior of the domain.