Project Report

Matteo Carlone, Fabio Conti, Alessandro Perri, Luca Predieri
Assignment of Artificial Intelligence for Robotics II

May 14, 2022

1 Introduction

The following report for the AIRo2 course will describe how we efficiently modelled an automated warehouse solution for transporting and relocating crates according to a company's needs. To implement the AI plan for the automation of the environment we took advantage of the PDDL+ planning language features . Thanks to Processes, Events, and Actions, we achieved every requirement the company asked for, including all the extra ones.

1.1 Assignment scenario

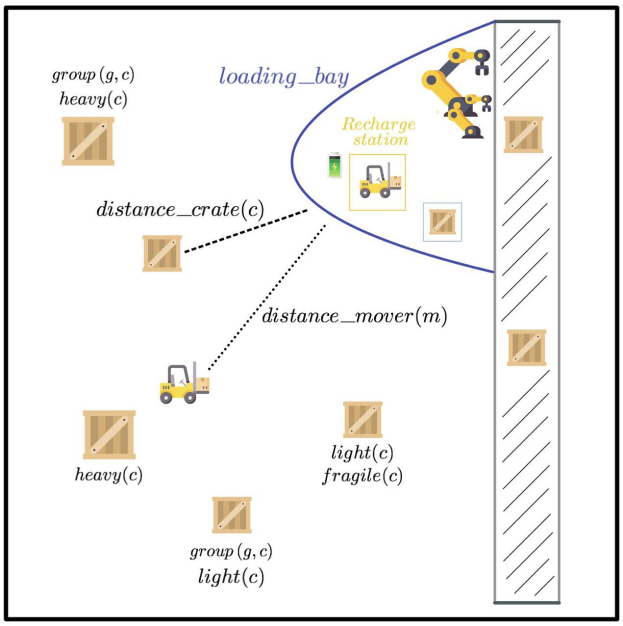


Figure 1: *Illustration of the assignment scenario*

The factory scenario comprehends the following agents:

- Two mobile robots (**movers**):
  Both capable of moving and transporting crates inside the whole warehouse structure.
- Two loading arms (**loaders**):
  Which are in charge of loading the crates on top of a conveyor belt.

The warehouse space is organized in two main sections:

- A **Loading-bay**:
  In this location the movers will either start a new task or recharge if needed. Moreover, the two loaders are firmly located here for picking up the crates and put them on the conveyor belt.
- A **Storage area**:
  This Location will contain all the crates at different distances from the loading-bay. The movers will pick up and transport these items to the loading-bay.

1.2 Installing and running

1.2.1 ENHSP

This project works with **ENHSP**, which stands for *Expressive Numeric Heuristic Planner*. It is a forward heuristic search planner, but it is expressive in that it can handle:

- Classical Planning
- Numeric Planning with linear and non-linear (!!) expressions
- Planning with discretised autonomous processes and events
- Global constraints, which are the analogous of always constraints of PDDL

The planner reads in input a PDDL domain and problem file,and it provides you with a plan (a sequence of actions, events, and processes). In the case of planning with processes, the plan is a time-stamped plan (associated to each action, you find the time at which that instance of the action has to be executed). In dealing with autonomous processes, ENHSP discretises the problem (with a $\delta = 1s$ by default); so the plan is guaranteed to be valid only with respect to that discretisation.

notes:

1. ENHSP supports PDDL 2.1 in particular, and PDDL+ (for the support of autonomous processes) and also events (only recently introduced).
2. The planner does not support *Durative Actions* (you need to compile them out) and is not stable with the whole Action Description Language.

1.2.2 Installing

The planner is written completely in JAVA, and in particular it works with JAVA 15. Before using the planner, make sure to have the Java machine installed on your computer. To use the planner on your own machine clone this GitLab repository.

1.2.3 Running

The planner can be executed from the root folder using the following command:

```
java -jar enhsp-dist/enhsp.jar -o <domain_file> -f <problem_file>
```

2 Brief description of the problem

When the warehouse receives new orders, the corresponding crates to be moved are communicated to the robots, that take them to the loading-bay where a dedicated robot can put them on the conveyor belt.

2.1 Movers details:

Depending on what the crates contain, we can distinguish **light** (weight lighter than 50kg) and **heavy** (weight higher than 50kg) crates, and **fragile**/**not fragile** crates depending on what kind of goods they contain. The way the movers deal with the carrying of the crates depends on their content and their **grouping**. If the current crate processed belongs to a certain group, the remaining ones of the same group have to be loaded subsequently. The movers can transport the crates in 3 different ways:

- **Transporting a light crate**: Only one mover transports one light crate.
- **Co-transporting a light crate**: Two movers transports one light crate.
- **Co-transporting a heavy crate**: Two movers transport an heavy crate.

If a crate contains a fragile good the movers must cooperate to transport it even if the weight is under the heavy-light threshold. Once the movers are running without a load their velocity are set to be $10dist - unit/time - unit$. If they are transporting any kind of crate the mover will change its velocity according to the crate's weight and the transporting modality. The following table summarizes the mover's velocity in the different conditions:

	Single mover Transport	**Coop - Transport**	**Fragile Transport**
Heavy	$NONE$	$v = 100/weight(crate)$	$v = 100/weight(crate)$
Light	$v = 100/weight(crate)$	$v = 150/weight(crate)$	$v = 100/weight(crate)$

Table 1: *Velocity of the movers in different condition of transport*

In addition to the different velocity constraint, every mover also holds a limited battery. The energy juice can be loaded up to 20 units. The single battery unit will expire every second the robot operates regardless of the transporting modality used. The movers can recharge only once they dock to the loading-bay.

2.2 Loaders details:

Since the beginning of the project we considered two **loading arms** to operate the placing of the crates onto the **conveyor belt** as the extra specification stated. As stated from the company the two arms have different capabilities when it comes to loading crates. The second loader is a cheaper one compared to the first one. It cannot load heavy crates. If any of the arms have to load a fragile crate the time required will change according to the following table:

	Main Arm	**Cheap Arm**
Heavy	$t = 4$ *time_units*	$NONE$
Heavy Fragile	$t = 6$ *time_units*	$NONE$
Light	$t = 4$ *time_units*	$t = 4$ *time_units*
Light Fragile	$t = 6$ *time_units*	$t = 6$ *time_units*

Table 2: *Time required by the loaders to load all types of crate onto the conveyor belt*

3 Structure of code

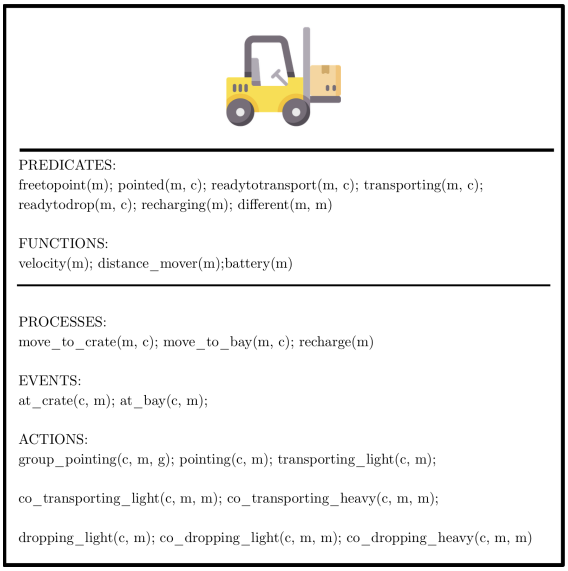
we divided the coding of the AI plan into two main tasks:

1. Bringing the crates collected in the storage area to the loading-bay (**movers' task**)
2. Loading the crates picked up from the loading-bay on the conveyor belt (**loaders' task**)

Such tasks are an high level description of the general problems the AI has to solve. To achieve such behaviour we organized these two points into smaller sub-problems, to satisfy all the proposed requirements while keeping an eye on all the given constraints.

3.1 Objects

Every **object** is associated with a set of predicates, functions, actions, processes, and events. These instances are fundamental for the description of every state of the final plan. The objects used to describe the warehouse environment are the following:



PREDICATES:
freetopoint(m); pointed(m, c); readytotransport(m, c); transporting(m, c); readytodrop(m, c); recharging(m); different(m, m)

FUNCTIONS:
velocity(m); distance_mover(m);battery(m)

PROCESSES:
move_to_crate(m, c); move_to_bay(m, c); recharge(m)

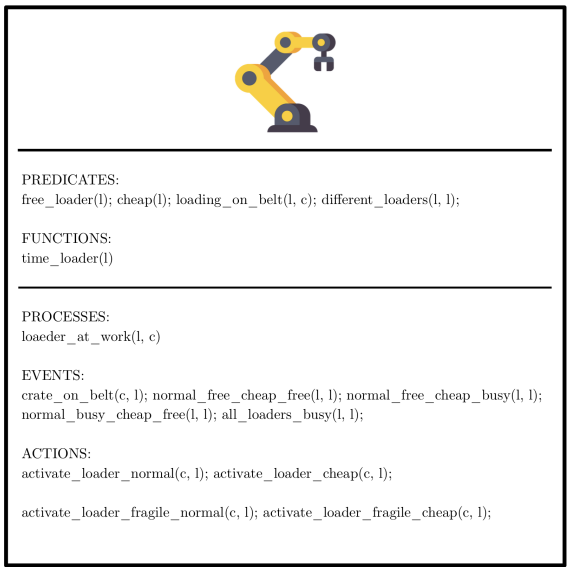
EVENTS:
at_crate(c, m); at_bay(c, m);

ACTIONS:
group_pointing(c, m, g); pointing(c, m); transporting_light(c, m);

co_transporting_light(c, m, m); co_transporting_heavy(c, m, m);

dropping_light(c, m); co_dropping_light(c, m, m); co_dropping_heavy(c, m, m)

(a) *Mover*



PREDICATES:
free_loader(l); cheap(l); loading_on_belt(l, c); different_loaders(l, l);

FUNCTIONS:
time_loader(l)

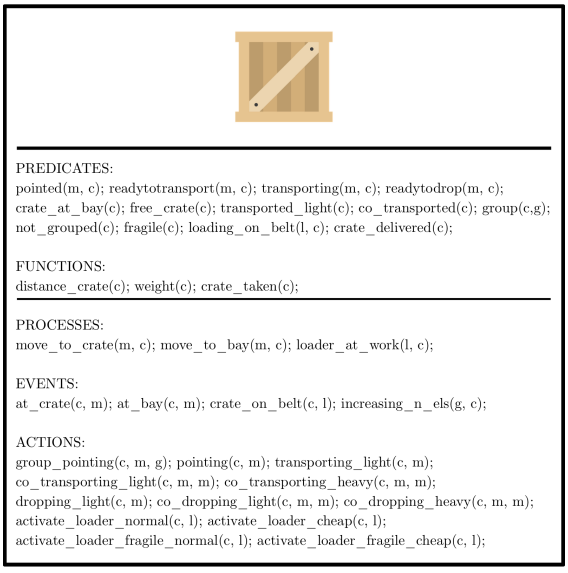
PROCESSES:
loaeder_at_work(l, c)

EVENTS:
crate_on_belt(c, l); normal_free_cheap_free(l, l); normal_free_cheap_busy(l, l); normal_busy_cheap_free(l, l); all_loaders_busy(l, l);

ACTIONS:
activate_loader_normal(c, l); activate_loader_cheap(c, l);

activate_loader_fragile_normal(c, l); activate_loader_fragile_cheap(c, l);

(b) *Loader*



PREDICATES:
pointed(m, c); readytotransport(m, c); transporting(m, c); readytodrop(m, c); crate_at_bay(c); free_crate(c); transported_light(c); co_transported(c); group(c,g); not_grouped(c); fragile(c); loading_on_belt(l, c); crate_delivered(c);

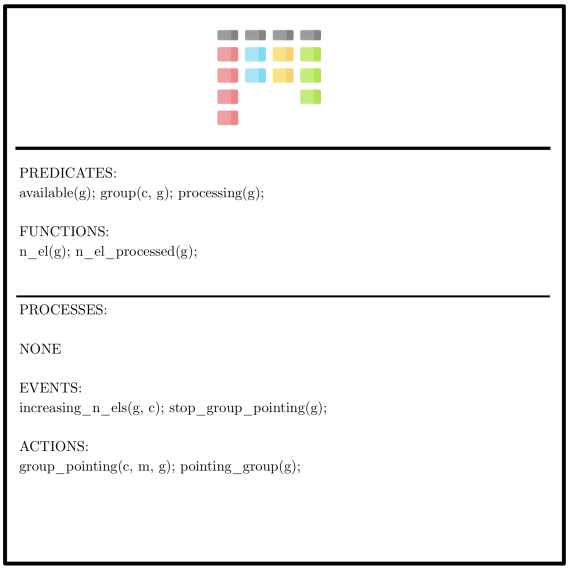
FUNCTIONS:
distance_crate(c); weight(c); crate_taken(c);

PROCESSES:
move_to_crate(m, c); move_to_bay(m, c); loader_at_work(l, c);

EVENTS:
at_crate(c, m); at_bay(c, m); crate_on_belt(c, l); increasing_n_els(g, c);

ACTIONS:
group_pointing(c, m, g); pointing(c, m); transporting_light(c, m); co_transporting_light(c, m, m); co_transporting_heavy(c, m, m); dropping_light(c, m); co_dropping_light(c, m, m); co_dropping_heavy(c, m, m); activate_loader_normal(c, l); activate_loader_cheap(c, l); activate_loader_fragile_normal(c, l); activate_loader_fragile_cheap(c, l);

(c) *Crate*



PREDICATES:
available(g); group(c, g); processing(g);

FUNCTIONS:
n_el(g); n_el_processed(g);

PROCESSES:

NONE

EVENTS:
increasing_n_els(g, c); stop_group_pointing(g);

ACTIONS:
group_pointing(c, m, g); pointing_group(g);

(d) *Group*

Figure 2: *List of predicates, functions, processes, events and actions related to the main objects of the domain*

3.2 Movers' point of view

The object *mover* is the main agent of the transporting side of the plan. The set of actions performed by the mover at every task follows always a similar pattern.

3.2.1 Pointing sequence

the first action the robot performs is to point a free crate(`free_crate(c)`). the pointing action is the method the movers exploit to decide which crate to reach. The pointing action depends on the grouped/not-grouped feature of the crates(`group(g, c)`, `not_grouped(c)`). These Two features will start two different actions routines for the movers to arrive at the crate's spot. If the planner decides to point a particular group((`pointing_group(g)`), the movers will focus their workflow on crates that belongs to the initially set group ((`group_pointing(c, m, g)`). If a not-grouped crate is pointed (`pointed(c, m)`) no grouping actions will be activated ((`group_pointing(c, m, g)`). To manage the grouping actions, some function are initialized at the beginning of every problem. The two functions are:

- The number of elements belonging to a certain group waiting to be processed (`n_el(g)`).
- The number of elements belonging to a certain group already processed (`n_el_processed(g)`).

Every time a mover points (`freetopoint(m)`) a grouped crate, `n_el_processed(g)` gets incremented by 1 unit. When `n_el_processed(g)` matches the number associated to `n_el(g)`, all the crates of the i-th group have been dropped in the bay. The planner is now free to point either another group or a non-grouped crate.

3.2.2 Crate approach

Once the mover points to a certain crate, it'll start moving towards the crate's location (`move_to_crate(m, c)`) with a constant velocity(`velocity(mover)`). This process will increase the distance of the mover relatively to the bay(`distance_mover(m)`). Once the mover's distance matches the distance of the crate to the bay (`distance_crate(c)`) the `at_crate(m, c)` event will stop the process of motion. This event will trigger the `readytotransport(m, c)` predicate. This instance will trigger one of the following transporting actions:

- `transporting_light(c, m)`: The mover loads a light crate `weight(c)` by itself and triggers the predicate to transport it back to the bay.
- `co_transporting_light(c, m)`: The mover performs the loading action on a light crate with the other mover. They will collaborate to place it back to the bay.
- `co_transporting_heavy(c, m)`: The mover performs the loading action on a heavy crate with the other mover. They will collaborate to place it back to the bay.

3.2.3 Bay approach and unloading of the crates

The execution of any of these actions will trigger the `transporting(c, m)` predicate. This predicate will start the process of moving the crate back to the bay (`move_to_bay(m, c)`). The velocity at which the mover will travel back to the bay depends on the previously mentioned table related to this topic. Once the robot's distance form the bay approaches 0, the (`at_bay(m, c)`) event will stop the process of moving back to the bay. This event will also activate the `ready_to_drop(c, m)` predicate. This instance will enable the dropping of the crate on the loading bay. The dropping actions directly depend on how the crate was transported to the loading bay in the first place. The following actions are the possible options for the unloading of the crate on the bay:

- `dropping_light(c, m)`: if the mover loaded a light crate `weight(c)` by itself, the crate will also be dropped without any kind of collaboration.
- `co_dropping_light(c, m)`: if the mover loaded a light crate with the other mover, they will also collaborate to drop it to the bay.
- `co_dropping_heavy(c, m)`: if the movers loaded an heavy crate, they will also collaborate to drop it to the bay.

3.2.4 Battery and recharging process

For all the time the mover navigates around the warehouse, it will lose its battery. The `battery(m)` function will either:

- decrease according to the previously stated conditions,
- or increase if the `recharge(m)` process is activated.

The recharge station of the mover is located at the loading-bay. The mover has to be docked at the bay to activate the charging process (`recharging(m)`). Since the movers won't move if the battery runs out, The planner has to think about the time the mover will take to reach the crate and bring it back to the bay according to the remaining battery. If by any chance the robot wouldn't be able to complete a task, due to a battery drain, the planner will not find any kind of plan to solve the given problem. (Ex.3)

3.3 Loaders' point of view

As previously mentioned, there are 2 different loading arms collaborating inside the loading-bay. The manipulators have not the same capabilities when it comes to loading crates on the conveyor belt. The cheaper version (`cheap(m)`) of the arm will not be able to operate heavier crates, while the default one can. The loading time doesn't change according to the capabilities of the loader. The only loading-time difference depends on the fragility of the crate's content.

3.3.1 Loading crates on the conveyor belt

The loading routine is exactly the same for every loading session regardless of the kind arm picking up the crates and the crates' content. The loading bay can only be filled with just one crate at a time. Some mutually exclusive events manage the activation and status of the two loaders during operation:

- `normal_free_cheap_free(l, l)`: both the loaders are ready to start a task.
- `normal_free_cheap_busy(l, l)`: the cheap loader is operating a task while the normal one is ready to pick up another crate.
- `normal_busy_cheap_free(l, l)`: the normal loader is operating a task while the cheap one is ready to pick up another crate.
- `all_loaders_busy(l, l)`: Both arms are taking care of a certain task.

The activation of any of those *states* will activate and deactivate the following Boolean predicates:

- `can_drop_light()`: which is a precondition for the mover to drop a light crate in the loading bay. The loaders will then be capable of picking up the crate (`loading_on_belt(c, l)`).
- `can_drop_heavy()`: which is a precondition for the movers to co-drop a heavy crate in the loading bay. The normal loader will then be capable of picking up the crate.

The activation and deactivating of these events is handled by the *event_i()* Boolean predicates. These instances will let the event be executed only for the instant the preconditions are met preventing them to repeat in time. There exist 4 different actions for the activation of the loading process:

- `activate_loader_normal(c, l)`: Once the mover will drop a non-fragile crate in the loading-bay (`crate_at_bay(c)`) and the normal loader will be free to start a new task (`free_loader(l)`), this action can take place and trigger the loading process (`loader_at_work(l, c)`). This action will also set the time of execution of the task to 4 time-units (`time_loader(l)`) according to the fragility of the crate.
- `activate_loader_cheap(c, l)`: This action will execute the same effect as the previous one but everything is referred to the cheap version of the arm.

- `activate_loader_fragile_normal(c, l)`: This action differs from the first one only by the fragile feature on the crate. Since the picked up crate is fragile, the execution time will be scaled up by 2 time-units, setting the operation time to 6 time-units.
- `activate_loader_fragile_cheap(c, l)`: this action acts the same way as the one above but it refers everything to the cheap arm.

The `crate_on_belt(c, l)` event will be executed at the expiration time of the loading process. This instance other than stopping the process will also set the `crate_delivered(c)` predicate to *true* satisfying the goal condition.

4 Performance Analysis of the planning engine

To evaluate the performances of the planning engine, we have run all the problems using different search strategies and heuristics. Specifically, we have launched the planner using all available options of *-planner*. Using this flag, you will be able to select a certain combination of search strategies and heuristics, for whom the level of support is known (you can find these info by checking the table of the high-level configurations here. All reported results were achieved by running the planner on a machine equipped with Apple M1 Pro 8-core CPU, 16 GB of memory (unified, ARM architecture), running MacOs Monterey 12.3.1. 1 GB of memory were made available for each run of every problem. We only had problems with **problem 4** with **opt-blind**, this planner option is pretty expensive in terms of computation, because the configuration is based on a blind heuristic that gives 1 to state where the goal is not satisfied and 0 to state where the goal is satisfied. We tried to dedicate up to 8196 MB of RAM to ENHSP but it wasn't enough.

The performances are reported here:

	Problem 1			Problem 2		
Data Evaluated High Level Configuration	Planning Time	Elapsed Time	Expanded Nodes	Planning Time	Elapsed Time	Expanded Nodes
sat-hmrp	914 ms	46.0	94	1042 ms	76.0	142
sat-hmrph	871 ms	41.0	95	1208 ms	75.0	996
sat-hmrphj	Java error			Java error		
sat-hadd	927 ms	30.0	346	1305 ms	54.0	3176
sat-hradd	958 ms	30.0	346	1315 ms	54.0	3176
sat-aibr	1225 ms	25.0	237	2005 ms	52.0	888
opt-hmax	1921 ms	23.0	42217	7007 ms	46.0	210662
opt-hrmax	2006 ms	23.0	42217	6824 ms	46.0	210662
opt-blind	1712 ms	23.0	89450	7386 ms	46.0	590569

Table 3: *Performances obtained by each high-level configuration for Problem 1 and Problem 2 ('-' indicates that the grounding process run out of memory)*

	Problem 3			Problem 4		
Data Evaluated High Level Configuration	Planning Time	Elapsed Time	Expanded Nodes	Planning Time	Elapsed Time	Expanded Nodes
sat-hmrp	Unsolvable			1264 ms	72.0	292
sat-hmrph	Unsolvable			1271 ms	74.0	249
sat-hmrphj	Unsolvable			Java error		
sat-hadd	Unsolvable			13403 ms	53.0	213432
sat-hradd	Unsolvable			13789 ms	53.0	213432
sat-aibr	Unsolvable			14013 ms	50.0	9034
opt-hmax	Unsolvable			191289 ms	44.0	3904651
opt-hrmax	Unsolvable			190387 ms	44.0	3904651
opt-blind	Unsolvable			-		

Table 4: *Performances obtained by each high-level configuration for Problem 3 and Problem 4 ('-' indicates that the grounding process run out of memory)*

As you can see in the tables above, there are two macro-types of high-level configuration, the first one has the prefix **sat-** and it is designed to work for sat planning. The search strategy used for this kind of

configurations is **Greedy Best First Search** (**GBFS**), except for *sat-aibr* which uses **A***. The latter is also used the second type of configurations, which starts with the prefix **opt-** and it is used for optimal planning. It is easy to understand that the former will take less time to find a plan than the latter and will also expand way less nodes, but the *Elapsed Time* for the returned plan will be generally much higher. For instance, in *Problem 1*, the optimal plan requires 23 [s] but for *sat-hmrp* configuration the plan provided requires 46 [s]. A good trade-off can be obtained using *sat-aibr* configuration, where the plan is only 2 [s] longer than the optimal one. This configuration results the most suitable also for *Problem 2* and *Problem 3*, since the plan provided is reasonably fast and either the *Planning time* and the *Expanded Nodes* are drastically lower than the available optimal plans. The *sat-aibr* configuration is trivially based on *Additive Interval-Based Relaxation*(**AIBR**), which can be seen as an over-approximation of **IBR** (i.e. the basis of most numeric planning heuristics) [1], and it is also the only one in sat planning which guarantees a full level of support of the features implemented by the language supported by **ENHSP**.
Please note that in *Problem 3* the mover doesn't have enough battery to reach one of the crates, even if it is at maximum charge. Therefore, the problem results unsolvable a priori.

References

[1] Scala, Enrico, et al. "Interval-based relaxation for general numeric planning." ECAI 2016. IOS Press, 2016. 655-663.