Visitor Management System
Cognitive Robotics Course - 097120
By Dr. Erez Karpas

Hadar Rosenwald 302905187
Shimon Sheiba 201110376
Shira Harel 302762075

github:
https://github.com/giliamit/escorting_project

video:
https://www.youtube.com/watch?v=Spp4gkNLneY&feature=youtu.be

This project in an enhancement of the team's final project implementing a visitor management and escorting system.
Our main goal of this work was to integrate both relatively complex PDDL skills learned during the course with a fully working simulated environment.
The original system was planned for one unit of the Turtlebot robot, and in this work, we combine multiple units in order to meet the goal of resource allocation.
We will first bring a detailed description of our PDDL domain and problem and start by discussing the objects and predicates:

We used 3 types of objects in our world; robot - representing by Turtlebot, person - the object needing escorting and waypoints – coordinates on a map which is a 2D representation of the world.

```
(:types
      robot
      waypoint
      person
)
```

And initialized as:

```
(:objects
   person1 person2 person3 - person
   robot1 robot2 - robot
   wp_dock1 wp_dock2 wp1 wp2 wp3 wp4 wp5 - waypoint
)
```

The visitor's flow within the system contains four steps:

1. Visitor (person) arrives.
2. A robot is allocated for this visitor who greets them personally and vocally in a specific waypoint on the map (a greeting point, which is the entrance of the building for example).
3. The person is being guided to a destination waypoint by the robot.

4. When guiding is completed, the robot informs the visitor vocally and moves to its next task.

The robot's initial mode is to be docked in its docker which is placed in a specific waypoint. This is represented by predicates:

(robot_at ?v - robot ?wp - waypoint)
(docked ?v - robot)
(dock_at ?wp - waypoint)
(greeting_waypoint ?wp - waypoint)

And its initializations:

(dock_at wp_dock1)
(dock_at wp_dock2)
(docked robot1)
(docked robot2)
(robot_at robot1 wp_dock1)
(robot_at robot2 wp_dock2)

The robot's battery level and its charging and consumption rates are functions which were taken in consideration.

(battery ?v - robot)
(consumption-rate ?v - robot)
(charging-rate ?v - robot)

They are initialized as:

(= (battery robot1) 100)
(= (battery robot2) 100)
(= (charging-rate robot1) 1.05)
(= (charging-rate robot2) 1.05)
(= (consumption-rate robot1) 0.9)
(= (consumption-rate robot2) 0.9)

With each robot's action, the battery level decreases by its consumption level, and when docking (will be discussed later) increases by its charging rate.

Upon person's arrival, these predicates become true:

(arrived ?p - person)
(person_at ?p - person ?wp - waypoint) – the visitor arrives at the greeting waypoint.

(arrived person1)
(arrived person2)
(arrived person3)

(person_at person1 wp1)

(person_at person2 wp1)
(person_at person3 wp1)

wp1 is decided to be the greeting waypoint.

To state that the robot currently can't perform any actions since it is currently docked, we included the (is_on ?v - robot) predicate. This is initialized as false.

Once a person has arrived, a robot is allocated to greet and guide them. To make sure that the same robot greets and guides that person, the (stick ?v - robot ?p - person) and (free_person ?p - person) predicates are used to create that mechanism.
This enables a situation for the planner to allocate the same robot to multiple visitors.

In order to prevent the robot from moving when in process of greeting and guiding a visitor, the (allowed_goto_waypoint ?v - robot) is in place.

We will now describe the actions flow logic using the discussed predicates.
Firstly, the goals for the plan are as follows:

```
(:goal (and
   (person_guided person1)
   (person_guided person2)
   (person_guided person3)
   (person_greeted person1)
   (person_greeted person2)
   (person_greeted person3)
   (person_at person1 wp2)
   (person_at person2 wp3)
   (person_at person3 wp4)
   (docked robot1)
   (docked robot2)
))
```

We chose to add a metric function that minimizes the sum of staying time of the visitors:

(:metric minimize (+ (total_stay person3) (+ (total_stay person1) (total_stay person2))))

For this purpose, we added a durative action stay that records the time between person's (arrived ?p) becomes true and (person_guided ?p) become true.
Each person has a (staying_rate ?p) parameter that indicates the person's priority. Meaning that a person with a higher priority has a higher staying rate. The point is to make the planner prefer a person with a higher priority.

We will present a scenario including one robot and one person.
**Whenever a goal state is reached it will be marked.

1. The person arrives at wp1 (greeting waypoint).

As mentioned before, the action stay starts and its duration is until the person (person_guided ?p) predicates turns true.

```
(:durative-action stay
        :parameters (?p - person)
        :duration (>= ?duration 0)
        :condition
                (and
                (at start (arrived ?p))
                (at end (person_guided ?p))
                )
        :effect (and
                (increase (total_stay ?p) (* #t (staying-rate)))
                (at start (staying ?p))
                )
)
```

2. Robot undocks using undock durative action.
   The undock action is a wrapper action. The duration of this action depends on the duration of all other actions that take place when the robot is undocked.
   During this action, the robot's battery decreases by the consumption rate given.
   This also makes the robot (is_on ?v) predicate true which enables it to move.

```
(:durative-action undock
        :parameters (?v - robot ?wp - waypoint)
        :duration (>= ?duration 0)
        :condition (and
                (at start (dock_at ?wp))
                (at start (docked ?v))
                )
        :effect (and
                (at start (not (docked ?v)))
                (at start (undocked ?v))
                (at end (not (undocked ?v)))
                (at start (is_on ?v))
                (at end (not (is_on ?v)))
                (decrease (battery ?v) (* #t (consumption-rate)))
                )
)
```

3. Robot moves from docking waypoint to greeting waypoint using the durative action goto_waypoint.
   This action (and also some other actions) requires the battery level to be positive. Also, (allowed_goto_waypoint ?v) needs to be true. At this point, this is enabled from initialization, and the action's affect turns this into False.
   The distances between all waypoints were calculated and are used to calculate the duration of the action.
   (:durative-action goto_waypoint

```
    :parameters (?v - robot ?from ?to - waypoint)
    :duration(= ?duration (/ (distance_x_y ?from ?to) (velocity ?v)))
    :condition (and
        (over all (undocked ?v))
        (at start (robot_at ?v ?from))
        (over all (general_waypoint ?from))
        (over all (general_waypoint ?to))
        (over all (allowed_goto_waypoint ?v))
        (over all (is_on ?v))
        (over all (>= (battery ?v) 0))
        )
    :effect (and
        (at start (not (robot_at ?v ?from)))
        (at end (robot_at ?v ?to))
        (at end (not (allowed_goto_waypoint ?v)))
        )
)
```

4. Once the robot has arrived at the greeting waypoint it can greet the person and perform the greet_person durative action. During this action, a vocal greeting message is played.
   As mention before, free_person and free_robot are required for this action to prevent a situation where a robot/person are already paired with another.
   In result of this action, the robot and person are paired together using (stick ?v ?p) and free_person and free_robot become false. Another result is (allowed_goto_waypoint ?v) being true again.
   (person_greeted ?p) is a condition that will be used in a future action and is a goal state.

```
(:durative-action greet_person
    :parameters (?v - robot ?wp - waypoint ?p - person)
    :duration (= ?duration 10) ;;audio file duration
    :condition (and
        (at start (undocked ?v))
        (at start (robot_at ?v ?wp))
        (at start (greeting_waypoint ?wp))
        (at start (arrived ?p))
        (at start (staying ?p))
        (at start (free_person ?p))
        (over all (is_on ?v))
        (over all (>= (battery ?v) 0))
        (over all (robot_at ?v ?wp))
        )
    :effect (and
        (at end (person_greeted ?p))
        (at end (allowed_goto_waypoint ?v))
        (at end (stick ?v ?p))
        (at start (not (free_person ?p)))
```

```
        )
)
```

5. Now that the person was greeted, they move to the destination waypoint as described in the goal state using the durative action goto_waypoint.

6. Once the robot has arrived at the destination waypoint, guide_person durative action takes place. During this action, a vocal goodbye message is played.
   As mentioned before, (person_greeted ?p) is a condition for guide_person to prevent a situation where a person that isn't greeted will be guided.
   As an effect of this action, the person is considered guided and **(person_guided ?p)** becomes true. Also, person location changes into the destination waypoint **(person_guided ?p)**.
   The robot and person become available again and (free_person ?p) and (free_robot ?p) turn true, (stick ?v ?p) and (allowed_goto_waypoint ?v) allows the robot to move to another waypoint and continue to its next task.

```
(:durative-action guide_person
        :parameters (?v - robot ?from ?to - waypoint ?p - person)
        :duration (= ?duration 6) ;;audio file duration
        :condition (and
                (at start (undocked ?v))
                (at start (robot_at ?v ?to))
                (at start (person_greeted ?p))
                (at start (destination_waypoint ?to))
                (over all (is_on ?v))
                (over all (>= (battery ?v) 0))
                (over all (robot_at ?v ?to))
                (over all (stick ?v ?p))
                )
        :effect (and
                (at end (person_guided ?p))
                (at end (allowed_goto_waypoint ?v))
                (at end (not (person_at ?p ?from)))
                (at end (person_at ?p ?to))
                (at end (not (stick ?v ?p)))
                (at end (free_person ?p))
                )


)
```

7. Lastly, to reach all plan goals, robot's (docked ?v) should turn true.
   To make this happen, another goto_waypoint action to the docking waypoint takes place.
   After this, the durative action dock is executed.
   The battery level increases in charging-rate continuously for an unfixed period of time, this is due to the condition for actions that require a positive level of battery.
   (:durative-action dock

```
        :parameters (?v - robot ?wp - waypoint)
        :duration (>= ?duration 0)
        :condition (and
                (over all (dock_at ?wp))
                (over all (robot_at ?v ?wp))
                (at start (undocked ?v))
                )
        :effect (and
                (at start (docked ?v))
                (at start (not (undocked ?v)))
                (at start (not (is_on ?v)))
                (increase (battery ?v) (* #t (charging-rate)))
                )
)
```

The second part of the project, we created a workspace which includes a 3D world and a 2D map (based on the 3D world). We used the Gazebo simulator to create the 3D world and to execute the plan visually and the RVIZ tool to map the 2D representation of the world.
The execution order was defined by a hierarchy of launch files that was responsible for:

1. Launching the Gazebo simulator with our world and localizing the robots in their initial positions.
2. Connecting between 3D simulator and 2D map for navigation purposes with all the relevant ROS topics and parameters.
3. Creating robots namespaces.
4. Loading the PDDL domain.

Using the rqt interface, the PDDL problem (represented by the turtlebot_explore.bash file) was loaded to the ROSPLAN knowledge base and the planning system was launched.
In order to interpret and execute actions according to the plan, we created a python script that subscribes to the kcl_ropslan/action_dispatch rostopic, parses the parameters for each different action in the plan, sends the necessary commands to the specific robot namespace topic and publishes the action status to the kcl_rosplan/action_feedback topic.

Since the rqt interface is not supporting the TIL (Timed initial Literals) and non-default Metric Function, while planning with rqt we used a slightly different PDDL problem and domain that was described earlier. The instruction of how to run the project will run the latter 'problem' and 'domain' (can view here https://github.com/giliamit/escorting_project/tree/master/src/ROSPlan/src/escorting/common) and they are supported by the rqt interface.
The PDDL description in this file is the fully theoretical solution that can be run directly through the popf3-clp planner. These domain and problem files can be found here: https://github.com/giliamit/escorting_project/tree/master/src/ROSPlan/src/escorting/theoretical_problem_domain

How to run our project?

get the files:
copy our workspace (git clone from https://github.com/giliamit/escorting_project)
directly to ~/catkin_ws_escorting/
cd ~/catkin_ws_escorting/src/ROSPlan/
run catkin_make
cd ~/catkin_ws_escorting/
run catkin_make

one terminal:
roslaunch ~/catkin_ws_escorting/src/ROSPlan/src/escorting/launch/main.launch
(wait for gazebo to launch)

second terminal:
rqt --standalone rosplan_rqt.dispatcher.ROSPlanDispatcher

third terminal:
source ~/catkin_ws_escorting/src/ROSPlan/src/escorting/turtlebot_explore.bash
(wait for all rosservice call to finish before returning to rqt again)

fourth terminal:
rostopic echo /kcl_rosplan/action_feedback

fifth terminal:
rostopic echo /kcl_rosplan/action_dispatch

sixth terminal (or the same one as the third with turtlebot_explore.bash)
python ~/catkin_ws_escorting/src/ROSPlan/src/escorting/listener_action_dispatch.py

once source turtlebot_explore.bash is done: in rqt window. hit 'plan' and enjoy (open speakers)