Online path planning

This lab will combine the concepts of occupancy grid mapping and path planning seen in previous sessions.
The main goal of this lab is to move a simulated or real Turtlebot-like robot from its current position to a goal position avoiding the obstacles in the environment.

Pre-lab

In this lab we are going to use the The Open Motion Planning Library to implement an on-line path planning algorithm. Therefore, the first we have to do is to install this library. Unfortunately, python bindings for OMPL have to be installed by source. Follow the instructions from https://ompl.kavrakilab.org/installation.html to install them. Follow the guide and use the following command:

```
./install-ompl-ubuntu.sh --python will install the latest release of OMPL with
Python bindings
```

it will take some time to finalize the installation.

Once the OMPL is installed in your PC, you should be able to run the notebook included in this package: ompl_example.ipynb. This notebook will show you how to use the OMPL library to implement basic path planning algorithm. Run it, understand the code, and complete whatever it is requested.

Online path planning architecture

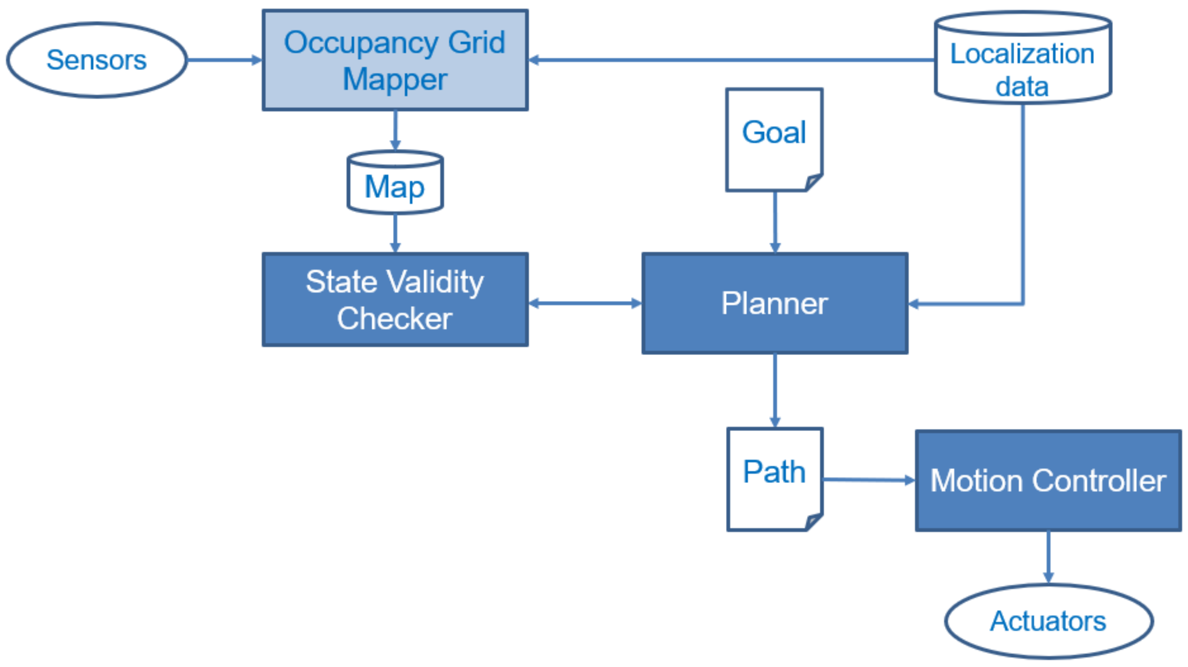
To implement a path planner that can run in an unknown environment in real time several modules are required:

- A map server that publishes the occupancy grid map of the environment that has been observer till this moment.
- A path planner that computes the path from the current position to the goal.
- A controller that moves the robot along the path.

All these modules have already been seen in previous labs or in the pre-lab:

- Map server: An occupancy grid map node has already been implementd and the Octomap server has already been tested.
- Path planner: Several path planning algorithms (i.e., A*, RRT, ...) have been implemented. Moreover, the Open Motion Planning Library (OMPL) can also be used to implement a path planner.
- Controller: A controller that moves a robot to a specific position has already been implemented in the `turtlesim` lab.

The following figure shows the relations between all these modules.



As you can see, the center piece is the `planner`, which is responsible for finding a path from the current position to the goal. The connection with the occupancy map is done through the `state validity checker` module that for each configuration examined by the `planner`, it check if there is a collision with the last available `map`. The `controller` takes the list of configurations computed by the `planner` to reach the goal (i.e., the `path`) and it is responsible for moving the robot along them. Every time that the `map server` publishes a new map, the `state validity checker` is also updated and the planned path has to be checked (i.e., new obstacles can make it invalid).

Implementation

Once introduced the main modules that have to be implemented, lets see in more detail how to implement each of them. It is very important to make each module ROS agnostic. It means that ROS specific code do not has to be included in the following modules. We will separate our code in a file containing all the functionalities of our modules and another file, a ROS node, that will make use of all these modules. This file (i.e., the ROS node) will contain all ROS related aspects: subscribe and publish all required messages, timers, configuration files, ...

State validity checker

The Octomap server publishes a 2D occupancy grid (`nav_msgs/OccupancyGrid`) map throug the topic `/projected_map` that is updated every time a new scan is received by the Octomap server. This 2D grid will be used by the `state validity checker` to check if a configuration is valid or not.

The `state validity checker` has to implement at least 3 functions: `set`, `isValid`, and `checkPath`.

`set(data, resolution, origin)`:

This function is called by the ROS node when a new occupancy grid is received. It is used to update the map that will be used to check the state validity.

- `data`: is a `int` array. To transform the `nav_msgs/OccupancyGrid/data` to a 2D matrix it has to be reshaped to `nav_msgs/OccupancyGrid/info/width x`

`nav_msgs/OccupancyGrid/info/height` and transposed
- `resolution` is the resolution of the map (meters per cell).
- `origin` is the position of the origin of the map (in meters).

`is_valid(x, y, distance=0)`:

This function checks if the position `(x, y)` defined in meters is valid or not. To do it, it is required to transform the position from meters to the cell index that the position belongs to. If `distance` is greater than 0, it is required to check if the cells at distance `distance` from the `(x, y)` position are also free or not. Take into account the map `resolution` to see how many cells around `(x, y)` have to be checked. The function returns `True` if all checked cells are free and `False` otherwise.

To transform from Cartesian position in meters ($Cartesian_p$) to cell index $Cell_p$, the following formula is used:

$$Cell_p = \frac{Cartesian_p - Map_{origin}}{Map_{resolution}}$$

If the requested position is outside the map, the function must return `True`. Otherwise, it will not be possible to send the robot to unknow positions. When the robot reaches a position closer to the unknown cells, the idea is that the map will be updated and the robot will known if the requested position is free or occupied.

`check_path(path, min_dist)`:

This function checks if a `path` generated by the path planner is still valid or not.

Once a `path` is computed, this function must be called every time that the `map` is updated. If the `path` becomes invalid, a new `path` must be computed.

While the path is just a list of configurations (i.e., x, y positions), the `checkPath` function must check not only that these positions are valid but also that the robot can move along them. A simple solution is to discretize the *segment* between two configurations using the `min_dist` parameter and check if each intermediate configuration is valid using the `isValid` function.

Path Planner

The path planner function needs a `start` and a `goal` position (i.e., `x, y` in meters) and access to the `state validity checker` to compute a valid `path` from the `start` to the `goal`. The `start` position can be the current position of the robot.

Only the `computePath` function must be implemented.

`compute_path(start, goal, state_validity_checker, dominion)`:

Using any path planning algorithm (i.e., A, *RRT, RRT*, ...) and the current `map`, this function computes a valid `path` from the `start` to the `goal`. The `path` is a list of `(x, y)` positions in meters.
To implement it, you can use the Open Motion Planning Library (OMPL) or any other path planning algorithm implemented by yourself.

The path planner will need access to the `state_validity_checker` while planning. It also will need to know the `dominion` of the environment. The `dominion` is a 2D array containing the minimum bound and maximum bound where the planner will search for a solution.

Controller

Once a `path` is computed, the controller is responsible for moving the robot along it. The controller only has to implement one function that must be called at a constant frequency if a valid `path` exists: `moveToPoint`.

`move_to_point(x, y, theta, x_goal, y_goal)`:

The `moveToPoint` function computes the desired forward velocity (ν_d) and desired angular velocity (w_d) that the robot must apply to move from the current position (x, y, θ) to the goal position (x_{goal}, y_goal). Because we want the robot to stay over the path as much as possible, the controller must compute first the desired angle between the current position and the goal (θ_d) and only when the desired angle is almost equal than the current one (i.e., $\theta_d \approx \theta$) move forward with a velocity proportional to the distance with respect to the goal. When the goal position is reached with some tolerance, this waypoint has to be removed from the `path` and the next configuration in the `path` will become the new goal.

$$\theta_d = \tan^{-1}\frac{y_g - y}{x_g - x}$$

$$w_d = K_h \text{wrap\_angle}(\theta_d - \theta)$$

$$\nu_d = \begin{cases} K_p\sqrt{(x_g - x)^2 + (y_g - y)^2} & \text{if } \theta_d \approx \theta \\ 0 & \text{otherwise} \end{cases}$$

Where $K_v > 0$ and $K_h > 0$.

Implementation

Following the skelethon provided in the file `utils_lib/online_planning.py`, implement a class for the `state_validity_checker` and the functions for the `path_planner` and the `controller` modules.

Complete the ROS node called `online_planning_node.py` that subscribes to the topics `/projected_map` from the Ocotmap server, `/odom` from the turtlebot3 pose, and `/move_base_simple/goal` published by the `rviz` visualizer. This node also has to publish the topics `/cmd_vel` to move the turtlebot3 as well as a `visualization_msgs/Marker` in the topic `path` to visualize the computed `path` in `rviz`.

The `online_planning_node.py` will import the functions previously defined and implement the following logics:

- When a `/projected_map` message is received:
    - The state validity checker must be updated using the `set` function.
    - If there is a `path` already computed, its validity must be checked using the `checkPath` function. If the `path` is not valid anymore, a new `path` must be computed to the same goal using the `computePath` function.
- Every time that a `/move_base_simple/goal` message is received the planner will be asked to plan a new `path` using the `computePath` function.
- At a constant frequency, e.g. at 10Hz, if there is a valid `path` computed, the `moveToPoint` function must be called with the current pose of the robot (i.e., x, y, θ obtained from the `/odom` topic) and the first configuration in the `path` as a goal. If this position is reached with some tolerance, this configuration must be removed from the `path` and the next configuration will become the new goal for the `moveToPoint` function. The ν_d and w_d

computed by the `moveToPoint` function have to be published as a `geometry_msgs/Twist` in the `cmd_vel` topic in order to move the turtlebot3. If there is no valid `path`, a `geometry_msgs/Twist` message with zero velocity must be published instead.

Deliverable

The `turtlebot_online_path_planning` package must be completed and compressed into a zip file. Do not forget to include in the README.md file:

- The name of the people in the group
- How to run your code or if some library must be installed
- If there is something special to explain.
- If there is anything that is not working properly.
- Several images or links to videos of the work done.