Lab 6: Path Planning

Learning Outcomes

- Understand Costmaps in ROS
- Implement the A* planning algorithm
- Utilize RViz for path visualization

Section 1: Requirements and Design

In this lab, you will implement the A* planning algorithm to develop a path for a robot to navigate a map.

Section 2: Installing the Lab

To perform this lab, you will need to get the lab6_path_planning template into your catkin workspace. First ensure that your Jet has internet access by connecting it using WiFi or ethernet. Next ssh into Jet and enter the following command:

```
wget http://instructor-url/lab6_path_planning/lab6_path_planning-code.zip
```

Where the url should be replaced by the URL provided by your instructor. Now unzip the lab:

```
unzip lab6_path_planning-code.zip -d ~/catkin_ws/src/jetlabs/lab6_path_planning
```

Delete the zip file:

```
rm lab6_path_planning-code.zip
```

To build the code, use the following command when you are in `~/catkin_ws/`:

```
catkin_make --pkg lab6_path_planning && source devel/setup.sh
```

To run the system, execute the following

```
roslaunch lab6_path_planning lab6_path.launch
```

To test the planner, execute the following

```
roslaunch lab6_path_planning path_test
```

Section 3: Survey of Navigation in ROS

Open the lab6 source code folder (`catkin_ws/jetlabs/lab6_path_planning/src`). First we will examine the maps that are used for navigation. In order to plan a path from the robot's current location to its goal, the robot must know the terrain. Robots can be supplied with a map at startup (as we do in this lab) or a robot can build its own map as it traverses its environment. Maps for navigation show regions that are safe to visit and regions that are impossible to access or would result in damage to the robot.

Copy the `maps/maze1.png` image to your host computer using `scp`. Open the image in a photo viewer, and examine the map. The white regions are "safe" and the black regions are inaccesible. ROS allows you to provide maps using png images. Open the `maps/maze1.yaml` configuration file. This file specifies the specifications for the map. The resolution means that each pixel in the map represents .05 square meters in the real world. The origin of the map aligns the map with the location of the robot's start location. The threshold values are used to specify what colors correspond to free and occupied space.

In this lab, we will implement a planning algorithm as a ROS plugin. Plugins are like ROS nodes, but they are not executed by themselves. Rather plugins are used by another node to add functionality. You will perform your implementation in `src/planner.cpp`. The settings for your plugin are contained in `global_planner_plugin.xml`.

In ROS there are two types of planning (global and local). Global planners use the entire map to develop a path that the robot can use to reach its goal. The local planner attempts to follow the plan created by the global planner while keeping the robot safe in cases where the map does not exactly replicate the environment.

Section 4: Using RViz

In this lab, we will use RViz to visualize the map and the plan. On the Jetson, launch lab6:

```
roslaunch lab6_path_planning lab6_path.launch
```

Then on the host computer run RViz:

```
rosrun rviz rviz
```

In RViz, select the 'Add' button and then select the 'By Topic' tab. Then highlight `move_base/global_costmap/costmap` and click okay. Now you can see the costmap. Notice that the map looks different from the original image. Ros inflates the inaccessible regions to ensure that the robot stays far enough away from these zones.

The goal of this lab is to plan a path that can take the robot from the origin of the map to the far opposite corner. We will test the functionality by running `rosrun lab6_path_planning path_test`. Try running this command. Now add `/test_plan` to the RViz window using the same process that you used to add the costmap. This plan is a placeholder for what we will implement in the next section.

Section 5: Implementing A*

Open `src/planner.cpp`. You can comment out the placeholder path creation code in the start of the `makePlan` method. If you need a reference for how the A* algorithm works, consult Wikipedia or RedBlobGames.

Below is a list of the essential functions for completing the lab:

`this->costmap.getSizeInCellsY()` returns height of the map in cells

`this->costmap.getSizeInCellsX()` returns the width of the map in cells

`this->costmap.getCost(int x, int y)` returns the cost of the cell at the x,y coordinates. When the cost is zero, the cell is safe. Non-zero cells are inaccessible.

`this->costmap.worldToMap(double wx, double wy, int& cx, int& cy)` converts the world coordinates (wx,wy) into cell coordinates (cx, cy).

`this->costmap.mapToWorld(int cx, int cy, double& wx, double& wy)` converts the map coordinates (cx, cy) into world coordinates (wx, wy).

`vector<Point> getNeighbors(Point p)` returns a list of the neighbors of the point. You must check that all of the neighboring points are safe.

Use `ROS_INFO()` to print variables for debugging.

Section 6: Challenge

Make your own map by creating an image with a photo-editing tool (Gimp is an excellent open-source option). Then create a yaml file with the configuration details for your map. Modify `launch/lab6_path.launch` to use your yaml file. You may want to update the goal pose in `src/path_test.cpp` to ensure that the goal is on a safe location in the map. Run your planning algorithm to see if you properly configured the map.