COMP-4510-project-2

Table of Contents

Running Files for task 3

In their own terminal run the following

```
$ roslaunch turtlebot3_gazebo turtlebot3_stage_4.launch
$ roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=gmapping
```

This should open a Gazebo and RviZ window.

Next, open the included RviZ config file

```
frontier-exploration / RvizProkectTwoConfig.rviz
```

It is important that the RviZ window config loads correctly,
the displays tab should contain,
- Map/local path/ global path from the start up files section (PART 1) - frontiers_map
- Subscribed to /frontiers_map - Displays the occupancy grid for the group of all frontiers - MarkerArray
- Subscribed to /visualization_marker_array
- Displays color coordinated dotes representing distinct clusters of frontiers, centroids and targets.
- EnergyMap
- Displays the paths found from robot to centroids created by expanding wavefront - This should initially be checked 'OFF' because this marker is distracting (but kinda cool)

Next, in a new terminal run,

```
$ roslaunch frontier-exploration turtlebot3_navigation.launch
```

If this command errors you may need to resource the terminal

```
$ cd catkin_ws
$ source devel/setup.bash
$ roslaunch frontier-exploration turtlebot3_navigation.launch
```

Next, run the frontiers identification script

```
$ rosrun frontier-exploration auto_exploration.py
```

(you may need to resource the terminal)
This will begin an automatic navigation software where the robot will find regions of unknown space and navigate to them in a way that optimally explores the entire map.

Running Additional Files Procedure (Not used for task 3)

See the position of the robot with,

```
$ rosrun tf tf_echo /map /base_footprint
```

Another way to display the position of the robot is with a listener script in terminal

```
$ rosrun frontier-exploration waffle_tf_listener.py
```

This will display the current position of the base of the robot with respect to the map.
Next, we want to issue a command to the robot to move.
This can be done with the moveActionClient.
Running

```
$ rosrun frontier-exploration moveActionClient.py -x <goal in x> -y <goal in y>
```

will translate the position of the robots base frame by (x,y) units.
Note: this translation is with respect to the robots frame not the map so for example translating (x=1,y=1) will move the robot right and up one unit. (instead of moving the coordinate (1,1) on the map.)
Here are a few examples to run.

```
$ rosrun frontier-exploration moveActionClient.py -x -1 -y 1
$ rosrun frontier-exploration moveActionClient.py -x -1 -y -1
```

You can also issue commands without specifying the argument, in the order (x,y)

```
$ rosrun frontier-exploration moveActionClient.py -1 1
$ rosrun frontier-exploration moveActionClient.py -1 -1
```

The pdf file frames.pdf displays a tf tree of the objects on the module.
To generate a new pdf, run,

```
$ cd catkin_ws
$ sudo apt install ros-noetic-tf2-tools
$ rosrun tf2_tools view_frames.py
```

Troubleshooting

- Cannot find package error
  - This is a problem that is liklely caused by the terminal not being sourced correctly. Which can be resourced by

  ```
  $ cd catkin_ws
  $ source devel/setup.bash
  ```

- Problems with tf package for view_frames.py
  - many of the original tf functions are depriciated and so tf2 has been used in exchange.

  ```
  $ sudo apt install ros-noetic-tf2-tools
  $ rosrun tf2_tools view_frames.py
  ```

- If the application is having trouble connecting to the robot try running the following to change the environment to use the waffle_pi robot. `console`
  ```
  $ export TURTLEBOT3_MODEL=waffle_pi
  ```

- If anything is added to the package, re-make the workspace `console $ catkin_make $ cd project/catkin_ws $ source devel/setup.bash`
- If you want to ensure a publisher is publishing `console rostopic echo <topic name>`

Summary

The following is instructions on running frontiers based auto-navigation software

- A frontier is an area where known unoccupied space meets unknown space.

The goal of this project is to construct a 'smart' navigation system that can detect frontier locations, segment them into distinct regions, then navigate / explore each region in a way that explores the entire map in the least amount of time.

The majority of the work here is done in auto_exploration.py

To detect and segment frontiers: - Procedure : 1) Subscribes to /map topic and takes an occupancy grid object
- when something is published on the /map topic an internal cache is updated

```
2) Detects the location of known obstacles and increases their
    size based on the cspace of the robot (discovered a priori).
                - This is done to prevent the algorithm from finding frontiers that
                    are to close to walls for the robot to explore.
                - Increasing the wall size is done with a standard morphological algorit

3) Detect frontier regions with edge detection
                - This is done with standard edge detection filtering
                    where an adjustable kernel is convoluted against
                    the occupancy grid to detect locations where known
                    unoccupied tiles (0) meets known space (-1).
                    Walls are excluded as edges.

4) Remove false regions and simplify frontiers topology
                - the edge detection method may mis-classify sensor errors as a new fron
                    This is a case where a small unknown region is completely surrounded
                    known unoccupied space. And seems to be most commonly caused by gaps
                    lidar sensors on the robot. (So something should be visible but isn'
                    likely become visible on the next time step.)
                - To remove these regions, each frontier is eroded based on the amount o
                        that surrounds it. (If a region is next to a large block of unkn
                        eroded. However, if a frontier is surrounded by known space, the
                        be reduced and in the event of a false frontier, be eroded into
                - After eroding, all remaining frontiers are dilated, (or have their are
```

and join frontier candidates that are extremely close but not co
over classification of frontier regions during connected compone

5) Frontier segmentation

- Here the list of all candidate frontier points are segmented into dist
- Features of a good segmentation can include, a continuous topological
  (or at least semi-continuous with small jumps.) distance from walls
  are reachable by the robot.
- Here connected component analysis is used to segment each frontier as
  topological regions.
  By semi-continuous we mean that the region is either fully conne
  components are 'small'.
  This is done by running breadth first search (BFS) sequentially on e
  candidate and connecting the visited points.
  Successors in the BFS for each point are found by the neighbors of t
  Neighbors are determined by overlaying an (n X n) kernel.

Then to rank frontier segments,
- The expected utility of a centroid is an approximation of how quickly
the entire map can be explored if that centroid is discovered next.

- Here we calculate the utility of a centroid (i,j) by the expected map
  entropy around (i,j) over the sum of the 'safest' path from the robot
  to the centroid.

  $$U(i,j) = H(i,j)/d(i,j)$$

- This rewards exploring paths with high entropy (uncertainty) but discourages
  exploration to points that are either far away to challenging for the robot
  to reach.

- Distance is calculated using expanding wavefront algorithm from the robot to e
  centroid where the magnitude of the gradient between points corresponds to t
  points distance from walls.

  - expanding wavefront returns a dictionary of paths for each centroid where
    path contains both the sum of weights along that path and the indices wi

- Map entropy of a frontier, F, is found using random sampling in monte carlo si

  - Define a fixed 2D region, R, to sample the map.

  - For frontier F, randomly select v points as anchors

  - For each anchor v_i, fix origin of region R at v_i

  - randomly sample map points, w_j within region R centered at v_i

- Sum and normalize the entropy for each random sample for each v_i.

After the centroids are ranked their habitability is determine and position adjusted