Lab 5: Path Planning

Due: Thursday, November 11th 11:59 pm EST

You are living on the desert planet of Tatooine, and you have been tasked with building a robot for the Resistance that can navigate from its initial location to its destination, whilst efficiently avoiding stormtrooper obstacles on the way. Thankfully, Cozmo came along with you when you flew to Tatooine! But you need to solve the robotic path planning problem for Cozmo.

The objective of this lab is to implement and test path planning capabilities for Cozmo, specifically the Rapidly-exploring Random Tree (RRT) algorithm. To facilitate easier debugging when coding your RRT, we have broken this lab into two parts - a simulated portion that will have an autograder, and a lab demonstration with Cozmo. Cozmo's goal during this demonstration will be to navigate to a target point, while accounting for obstacles in real-time.

Part 1 – Simulation

For the simulation component, we have provided you with the following files to assist with development and debugging:

`utils.py`: contains general methods you may find of use, as well as the definition of the Node class.

`cmap.py`: representation of the map and c-space. Features include start location, goal location, obstacles, and path storage. Map configuration is loaded from the JSON file (located in `maps` folder) supplied at object creation.

`gui.py`: the visualizer

`autograder.py`: used to grade the RRT solution in simulation

`rrt.py`: implementation of RRT path planning

Step 1: Complete the `node_generator` method in `rrt.py`. This method should return a randomly generated node, uniformly distributed within the map boundaries. Make sure to check that the node is in free-space. Additionally, implement your code such that with a 5% chance the goal location is returned instead of a random sample.

Step 2: Complete the `step_from_to` method in `rrt.py`. This method takes as input two nodes and a limit parameter. If the distance between the nodes is less than the limit, you return the second node. Otherwise, return a node up to the limit distance away from the first node along the path to the second node.

Step 3: Complete the `get_global_node` method in `rrt.py`, which is a helper function used by `detect_cube_and_update_cmap` function. This method takes as input the angle (in radians) and origin of the local coordinate frame in terms of the global frame, as well as a node (x,y) coordinate in this local frame, and returns the node in the global frame. Remember from previous lectures that a point transformed from one frame to another follows the formula below, where θ and t_x and t_y would correspond to the angle and origin of the local coordinate frame in terms of the global coordinate frame, and p_x and p_y are the node (x,y) coordinates in this local frame:

$$ {}^3p = {}^3T_1 \, {}^1p = \begin{bmatrix} {}^3R_1 & {}^3t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} {}^1p \\ 1 \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta & {}^3t_x \\ sin\theta & cos\theta & {}^3t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^1p_x \\ {}^1p_y \\ 1 \end{bmatrix} $$

Step 4: Complete the `RRT` method in `rrt.py`. Complete the main loop of the algorithm by generating random nodes and assembling them into a tree in accordance with the algorithm. Code for goal detection, tracking parents, and generating the path between the start and end nodes is already provided within cmap.py.

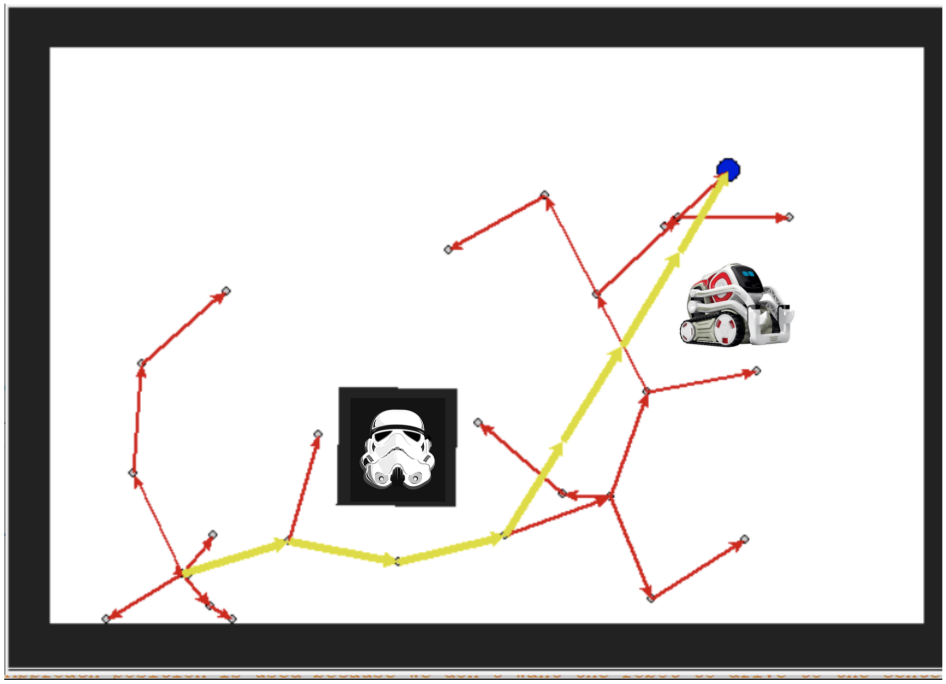
Step 5: Once the above steps are complete, validate that your RRT algorithm works. We have provided several maps for testing purposes, located in the `maps` folder. You can run the algorithm on one map with a graphical visualizer by executing `python rrt.py` (you can change the map in the main method at the bottom of the file), or you can run the algorithm on multiple maps at once without the visualizer by executing `python autograder.py gradecase.json`.

Step 6: Now RRT should be working, but it would return very convoluted paths. Improve its performance by implementing path smoothing in the `get_smooth_path` method in `cmap.py`.

Note that you may edit provided functions such as `detect_cube_and_update_cmap` if you would like to, but shouldn't need to.

Part 2 – Cozmo

Now it's time to deploy your RRT path planning algorithm on Cozmo, so he can avoid all those stormtroopers on the way! For this, you will need to complete the `CozmoPlanning` method in `rrt.py`. This method is executed by the `RobotThread` and should contain all your Cozmo behavior. Your implementation should use RRT to find a path to a specific predetermined target point, follow the path found by RRT, and re-plan (which means you need to reset and run RRT again) to avoid an obstacle cube that is added during navigation. But once a cube is placed down and it becomes visible to Cozmo, you can assume that it will not move. Note that the `detect_cube_and_update_cmap` method has been provided in `rrt.py`. Also note you are **NOT** allowed to use the go_to_pose() function as it does its own path planning.

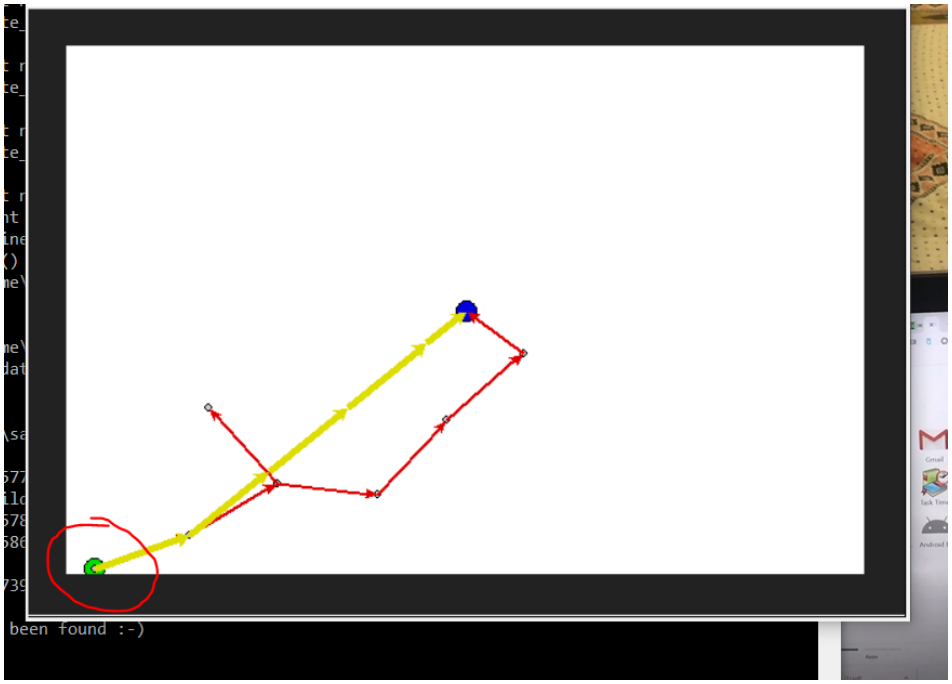


An example view of the RRT path generated around a cube obstacle. The stormtrooper is hiding within those cubes, so Cozmo must avoid it at all costs!

To test your code with the robot, run `python3 rrt.py -robot`. Cozmo's starting position (in mm) should be (50, 35, 0) ~ (x, y, theta) in the code/simulated arena as circled in red in the figure below, so it's very close to the bottom left corner of the arena. The destination position should be the center of the arena (map_width*1/2, map_width*1/2, any) ~ (x, y, theta), and the robot should roughly reach it. You are not required to use the arena, but if you don't use it, please make sure to mark where the center of the arena would theoretically be.

For the demo, we would like to see two things, namely: (1) the robot is following the RRT path, and (2) the robot detects the cube and goes around it. Here is an example video is, with the center of the arena marked by a white piece of paper: https://drive.google.com/file/d/18ki6tRPWwZgq8ZfLK86zrROzwERW0pxt/view?usp=sharing.

1) The steps of the demo are:
    a. You run the `python3 rrt.py -robot` command, with no cube obstacle. The robot starts following the path.
    b. You place an obstacle cube in the robot in the robot path. When the robot encounters the cube, the robot re-plans its RRT path so that it will go around the obstacle.r
    c. Robot roughly reaches the center of the arena, or if you didn't use an arena, it reaches a marking indicating the center of the arena.



The starting point of Cozmo (at the bottom left corner) has been circled in red. As Cozmo moves along the path, you must place a cube in the path. Note that the cube might not be detected unless the robot is directly facing the cube.

Grading: A portion of your grade will come from the autograder running your code on new maps in simulation. The remainder of your grade will be based on your demo. The exact point breakdown is as follows:

Part 1 – Simulation:	
RRT, autograded with 6 maps, 10 points per solved map	60 pts
Part 2 – Cozmo: Demo!	
The robot roughly follows the path found by RRT	10 pts
The robot roughly reaches the target point	5 pts
The robot follows a smoothened path	5 pts
The robot re-plans to avoid obstacle cubes	15 pts
The map is updated to reflect newly seen cubes	5 pts

Submission: Create a zip file that includes, `rrt.py`, `cmap.py` with you and your partner's names at the top of each file. Make sure the files remain compatible with the autograder. Only one partner should upload the file to Gradescope. If you relied significantly on any external resources to complete the lab, please reference these in the submission comments.

Good luck, and may the Force be with you! :)