HW 3—Sampling-based motion planning & inspection planning

1 General guideline

This homework will continue our study of sampling-based motion planning (in 2D as well as in 3D - in simulation and in the lab) and cover the topic of inspection planning. Writeups must be submitted as a PDF. LaTeX is preferred, but other typesetting methods are acceptable. Code for the programming component must be submitted in a zip archive. Plots generated as part of the programming component should be included in the writeup. The homework can be submitted in pairs.

Submission date is 20.1.2022 end of day.

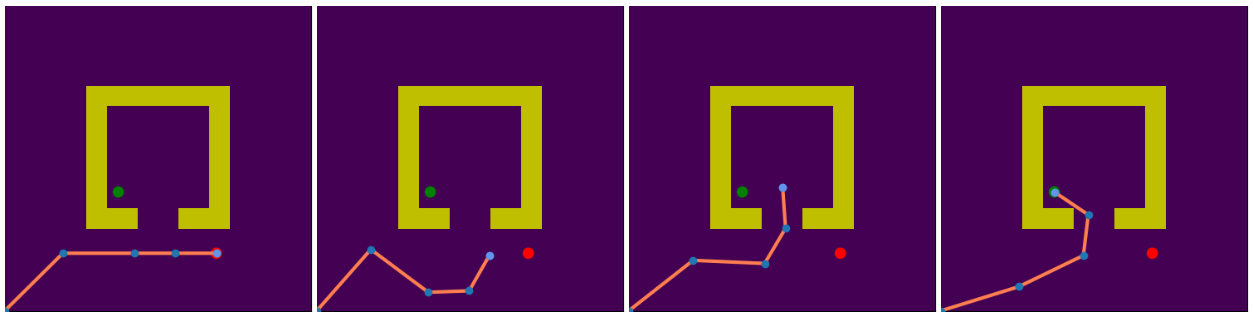
2 2D: A*, RRT and Inspection Planning

2.1 A* Implementation (30 points)

You will be implementing the weighted version of A* where the heuristic is weighted by a factor of ε. Setting $\varepsilon = 1$ gives vanilla A*. In order to help you, we have created a `MapDotEnvironment` class. Which is a simple environment for testing your motion-planning algorithms, so you just have to fill in the logic in `AStarPlanner.py` and `dot_building_blocks.py` files. Pay attention that in `run.py` you have the functions for running all the different examples (including two options for maps for the `MapDotEnvironment`).

1. Use an 8-connected neighborhood structure so that diagonal actions are also allowed. Each action has a cost equal to the length of the action i.e., cost of action $(dx, dy) = \sqrt{dx^2 + dy^2}$.
2. Use the Euclidean distance from the goal as the heuristic function.
3. Try out different values of ε to see how the behavior changes. **Report** the final cost of the path and the number of states expanded for $\varepsilon = 1,\ 10,\ 20$.
4. **Discuss** the effect of ε on the solution quality.
5. **Visualize** the final path in each case and the states visited (notice that the visualization automatically draw the states that were visited by extracting `expanded_nodes` so make sure to fill this list).

Note: The workspace is considered to be a continuous domain but the environment is discretized into grid cells. Thus, to test if a point (x, y) is in collision, we first find its corresponding cell and test if that cell is obstacle-free or not.



2.2 Motion Planning (25 points)

In the previous assignment, you implemented the basic building blocks for implementing sampling-based motion planning algoithms on the 2d manipulator environment. Here, we will use these to implement the single-query algorithms. You will start by implementing the RRT algorithm: at first, use the DotEnvironment (as in previous section) for debugging and then run and **report** results for the 2D robotic manipulator. In the file `RRTMotionPlanner.py`, implement the following functions:

1. `extend` - You will implement two versions of the extend function:

   E1 extending an edge from the nearest neighbor in the RRT tree all the way to the sampled configuration.

   E2 extending an edge from the nearest neighbor in the RRT tree by a step-size η towards the sampled configuration. Pick a small η of your choice and mention it in your write-up.

   For each version of the extend function **report** the performance (cost, time) and include a figure showing the final state of the tree for both biasing parameters. Which extend strategy would you employ in practice?

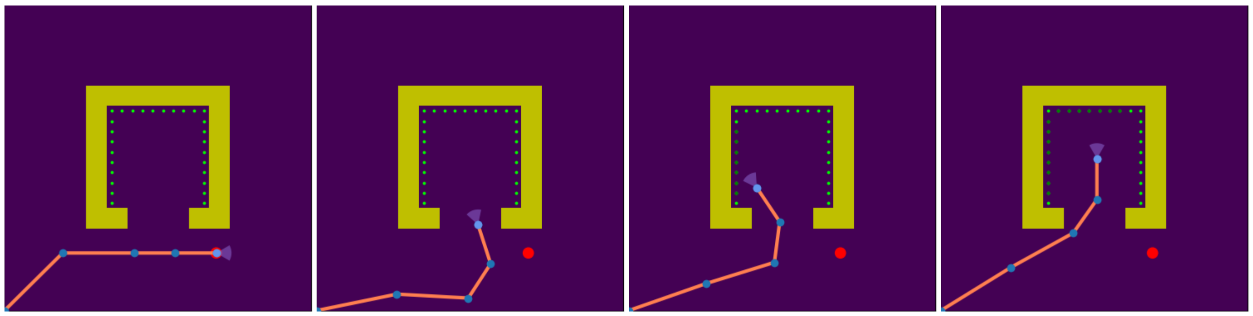
2. `compute_cost` - Compute the cost of the given path, as the sum of the distances of all steps.

3. `plan` - The body of your planning algorithm with goal biasing. Make sure that the output is a numpy array of the calculated path in the configuration space. This array should (i) include the start and target configuration and (ii) be of dimension $N \times 4$ for N configurations.

Report the performance of your algorithm (cost of the path and execution time) for goal biasing of 5% and 20% averaged over 10 executions for each, and attach the visualizations created by `visualize_plan` for one representative instance for each parameter combination.

2.3 Inspection Planning (25 points)

In inspection planning, we are require to inspect a set of points of interest (POI) and not reach a predefined goal. In this exercise you are required to adapt your RRT implementation to handle the



inspection task (this is slightly easier than the IRIS algorithm that will be taught in class). A vertex v_i in the RRT tree, will now also include the subset of POI that were seen so far when following the path from the root of the RRT tree to the configuration associated with the vertex and set $I_i = \{p_1, \ldots, p_i\}$ to be this subset of POI. Namely, if in the (vanilla) motion-planning problem we defined a vertex as $v_i := [\theta_{i,1}, \theta_{i,2}, \theta_{i,3}, \theta_{i,4}]$, it will now be $v_i := [\theta_{i,1}, \theta_{i,2}, \theta_{i,3}, \theta_{i,4}, I_i]$. Moreover, if the edge (v_{i-1}, v_i) exists, then the set of POI inspected must be $I_i = I_{i-1} \cup S(v_i)$, where $S(v_i)$ is the set of inspection points that the robot can see at step vertex v_i.

Next, we define the *coverage* of a vertex v_i as $\frac{|I_i|}{|I_{\max}|}$, where $I_{\max}$ is the set of all POI. Namely, a vertex with coverage of 1.0 corresponds to a path along which all POI have been inspected.

To this end, the stopping condition of your planner will be reaching not a goal vertex but a vertex with a desired coverage and return the path to it. Implement the following in `RRTInspectionPlanning.py`:

1. `extend` and `compute_cost` - These can stay the same as in the motion-planning task.
2. `compute_union_of_points` (in `building_blocks.py`) - Compute the union of two sets of inspection points.
3. `plan` - The body of your planning algorithm. Make sure that the output is a numpy array of the calculated path in the configuration space (should be a shape of $(N, 4)$ for N configurations, including start and end configurations).

Report your performance (cost of the path and execution time) for coverage of 0.5 and 0.75, averaged over 10 executions for each, and attach the visualizations created by `visualize_plan` for one representative instance for each parameter combination.
Discuss the changes you were required to do in order to compute inspection planning. Notice that for higher coverage the search may take a few minutes, so think about how you can improve your search! Make sure it is as efficient as possible.
Hint: In motion planning we biased the sampling towards a goal vertex. If there is no goal, how can we bias the sampling to improve coverage?

2.4 Competition (Bonus to final grade—up to 5 points)

In this exercise, you will also have the chance to compete in a short inspection competition. You are encouraged to improve your algorithm to collect more inspection points in less time. If you want to improve given modules, such as collision detection or the sensor function, you are allowed to, but make sure to keep your favorite edge extension mode as default. Notice that you will be tested on maps that are not provided with the code.

If you want to participate, all you need to do is to **notify** it in your report, and **make sure** that your code supports the `run_2d_rrt_inspection_planning` function in `run.py`:

The extra points will be credited to your final grade. First, second and third places will receive 5,3 and 1 additional points, respectively.

3 RRT*

3.1 2D RRT* (15 points)

Now we would like you to implement the RRT* algorithm for the 2D manipulator environment. Please fill in the logic in `RRTStarPlanner.py`

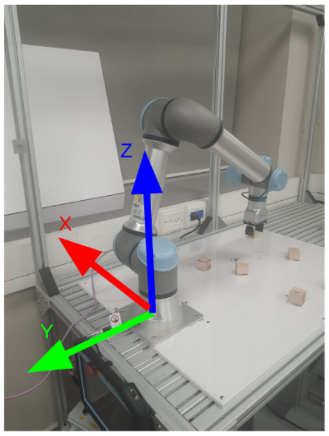
Report your performance (cost of the path and execution time) for coverage of 0.5 and 0.75, averaged over 10 executions for each, and attach the visualizations created by `visualize_plan` for one representative instance for each parameter combination.

Discuss the differences by comparing the algorithm to A* and RRT. Is one run sufficient enough for evaluating such algorithms? You are encouraged to run more different start and goal configurations.

3.2 3D RRT* (15 points)

We will now continue to the 3D environment and run the RRT* algorithm on it. You will study the effects of the parameters on the algorithm's performance. Then, we will visualize the paths in an advanced simulation and at the lab.

1. In this lab, there is the added constraint of ensuring that the robot does not hit the window. One solution could have been to model the wall as an obstacle approximated by a set of spheres. This is a bad idea (why?). Instead, add a condition to the function config_validity_checker()() to return `False` if the manipulator exceeds the plain 0.4 [m] in x-direction. Provide the function the configuration [130,-70, 90, -90, -90, 0][deg] (convert degrees to radians using the numpy.deg2rad() function) to verify that it indeed returns `False`.



2. Run the **RRTStarPlanner** class in RRTStarPlanner.py.

3. Find paths for each test as described below. **Report** the performance in four figures, one for each `p_bias` value, plotting the cost as a function of computation time for different `max_step_size` values and the success rate as a function of computation time for different `max_step_size` values. Do not terminate the planner once a first solution is found but continue running it to see if and how much the path quality is improved. Discuss how the performance (cost and time) of the planner is affected by the hyper-parameters `max_step_size`, `p_bias`. Since RRT* is non-deterministic you should provide statistical results (average over 20 runs for each test).[1]

   - Set `env_idx=2` in the Environment constructor.
   - Limit the maximum iteration to 2000 for each run.
   - Set start position to [110,-70, 90, -90, -90, 0][deg] and goal position to [50, -80, 90, -90, -90, 0][deg]

   run the tests for `max_step_size` = [0.05, 0.075, 0.1, 0.125, 0.2, 0.25,0.3,0.4] and `p_bias` = [0.05, 0.2]

4 Real World Basic Experience (20 points)

1. Choose the path with the lowest cost between all paths computed and bring it to the lab during the dedicated meeting (12/1 and 19/1, details to come). Together with the instructor, run it on the UR5e manipulator. include a video that visualizes it.

2. Watch the path computed by OMPL for the same path-planning problem.

Good luck!

[1]Remember to convert degrees to radians using the `numpy.deg2rad()` function.