HW 2—Sampling-based motion planning

1 General guideline

This homework will cover the topic of sampling-based motion planning. It includes two "dry" exercise (2 and 3) and one "wet" programming exercise (4). Writeups must be typed and 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.

2 Distribution of points in high-dimensional spaces (10 points)

In this exercise we will gain intuition on the behavior of how points are distributed in high-dimensional spaces. Throughout this exercise, think of the implication to sampling-based planners and especially asymptotically-optimal ones. You will need to use the formula for the volume of a d-dimensional unit ball (ball of radius one that resides in a d-dimensional Euclidean space). See `https://en.wikipedia.org/wiki/Volume_of_an_n-ball`.

Warmup (0) points

Please use your intuition only (and be honest ;-) to answer the following questions:

1. In a 2-dimensional unit ball (namely, a disk), how much of the volume is located at most 0.1 units from the surface (see Fig. 1)?
   (a) Roughly 1%
   (b) Roughly 10%
   (c) Roughly 20%
   (d) Roughly 60%

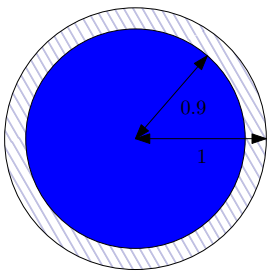

Figure 1: A 2-dimensional unit ball (blue) where the volume located at most 0.1 units from the surface is depicted in dashed lines.

2. In a 9-dimensional unit ball, how much of the volume is located at most 0.1 units from the surface?

   (a) Roughly 1%

   (b) Roughly 10%

   (c) Roughly 20%

   (d) Roughly 60%

Exercise (10 points)

Define the fraction of the volume that is ε distance from the surface of a d-dimensional unit ball as $\eta_d(\varepsilon)$.

Plot $\eta_d(\varepsilon)$ as a function of d for $\varepsilon = 0.2, 0.1, 0.01$ for $d = 2 \dots 10$.

Discuss the implications to reducing the connection radius required for a sampling-based algorithm to maintain asymptotic optimality.

3 Tethered robots (20 points)

We consider a 2D point robot translating amidst polygonal obstacles while being anchored by a tether to a given base point p_b. The robot may drive over the cable, which is a flexible and stretchable elastic band remaining taut at all times. We study the problem of constructing a data structure that allows to efficiently compute the shortest path of the robot between any two given points p_s, p_t while satisfying the constraint that the tether can extend to length at most L from the base.

1. (**2 points**) describe the structure of a path from a start to target configuration (robot location + tether description).

2. (**2 points**) Suggest an efficient way to encode the tether's description. Note that the robot can be at the same location but with completely different tether descriptions (e.g., circling once or twice around an obstacle) and what we need to define is the homotopy class of the tether.

3. (**6 points**) The *homotopy-augmented graph* G_h of a graph $G = (V, E)$ encodes for each vertex of G, all homotopy classes that can be used to reach the vertex using a tether of length L. Describe an approach to compute the homotopy-augmented graph of a given graph using a Dijkstra-like algorithm. Describe the nodes, how they are extended and when the algorithm terminates.

4. (**10 points**) Now we will now use the notion of a homotopy-augmented graph to efficiently answer queries solving the motion-planning problem for a point tethered robot.
   A query is given in the form of two points p_s, p_t and the h-invariant w_s describing the tethered placement at p_s. In order to compute a path (if one exists) between p_s and p_t with an original tether placement defined by w_s, we need to traverse the homotopy-augmented graph G_h^{vis} (the homotopy-augmented graph of the visibility graph).

- What new vertices do we need to add the homotopy-augmented graph G_h^{vis} to account for p_s and p_t?
- What new edges do we need to add the homotopy-augmented graph G_h^{vis} to account for p_s and p_t?
- How can we use the newly-constructed graph to efficiently solve our motion planning problem?

4 Motion Planning: Search and Sampling (70 points)

In this section you will be required to implement in Python different motion planning algorithms and study the parameters that govern their behaviors.

4.1 Code Overview

The starter code is written in Python and depends on numpy, matplotlib, imageio and shapely. We recommend you to work with virtual environments for python. If any of the packages are missing in your python environment, please use the relevant command:

```
pip install numpy
pip install matplotlib
pip install imageio
pip install Shapely
```

You are provided with the following files:

- `run.py` - Contains the main function. Note the command-line arguments that you can provide.
- `MapEnvironment.py` - Environment-Specific functions. Some of them have to be filled in by you.
- `map1/2.json` - JSON files describing the maps that the code will generate. They contain map boundaries, polygonal obstacles, and start and goal locations.
- `AStarPlanner.py` - A* Planner. Logic to be filled in by you.
- `RRTPlanner.py` - RRT planner. Logic to be filled in by you.
- `RRTStarPlanner.py` - RRT* planner. Logic to be filled in by you.
- `RRTTree.py` - Contains RRT tree code to be used by your implementations of RRT and RRT*.

The following are examples of how to run the code using different maps and planners:

```
python run.py –map map1.json –planner rrt –ext_mode E2 –goal_prob 0.05
python run.py –map map2.json –planner astar
python run.py –map map2.json –planner rrtstar –ext_mode E1 –goal_prob 0.05 –k 5
```

4.2 Environment Modelling

The planning consists of a 2D map. You have been provided with two maps `map1.json` and `map2.json`. You can use the former to *test* your implementation but report all results on the latter. Make sure to report your results on the provided start and goal positions. Note that environment-specific functions, such as collision detection and distance computation, are already implemented in the `MapEnvironment.py` file. In case of search algorithms like A*, the environment is considered to be a discrete grid while in sampling-based techniques the environment is assumed to be continuous. However, in this case since the underlying world is given to be grid, you can snap any continuous sample points onto the grid.

Note: During the whole assignment, use $< x, y >$ coordinate convention (i.e., <horizontal, vertical>) to describe 2D points. The plotting function will work properly as long you follow this convention.

4.3 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*. The main algorithm is to be implemented in `AStarPlanner.py` and `MapEnvironment.py` files.

1. Use an 8-connected neighbourhood 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).

4.4 RRT and RRT* Implementation (50 points)

You will be implementing a Rapidly-Exploring Random Tree (RRT) for the same 2D world. The main algorithms are to be implemented in `RRTPlanner.py` and `RRTStarPlanner.py` files. Note that since these methods are non-deterministic, you'd need to provide statistical results (averages over 10 runs).

1. Bias the sampling to pick the goal with 5%, 20% probability. **Report** the performance (cost, time) and include figures showing the final state of the tree for both values.

2. For this assignment, you can assume the point robot to be able to move in arbitrarily any direction i.e. you can extend states via a straight line. You will implement two versions of the extend function:

   E1 the nearest neighbor tries to extend all the way till the sampled point.

   E2 the nearest neighbor tries to extend to the sampled point only by a step-size η. Pick a small η of your choice and mention it in your write-up.

   As before, 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?

3. You will also be implementing RRT* as part of this question. You can implement this on top of your RRT planner with consideration for rewiring the tree whenever necessary. Choose k the number of nearest neighbors to connect each node as (i) a constant (choose several values) and (ii) $k = O(\log n)$, where n is the number of nodes in the tree. For each version, **plot** (1) the success rate to find a solution as a function of time[1] and (2) the quality of the solution obtained as a function of time for one representative run (with your choice of goal bias and extend functions).

[1]The success rate at time t is the portion of runs that found a solution by time t. Note that this graph should hit one at some point because RRT and RRT* are probabilistically complete.