HW 2—Sampling-based motion planning

1 General guideline

This homework will cover the topic of sampling-based motion planning. It includes one "dry" exercise (**??**) and one "wet" programming exercise (**??**). 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 (20 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 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. **??**)?
   (a) Roughly 1%
   (b) Roughly 10%
   (c) Roughly 20%
   (d) Roughly 60%
2. In a 9-dimensional unit ball, how much of the volume is located at most 0.1 units from the surface?

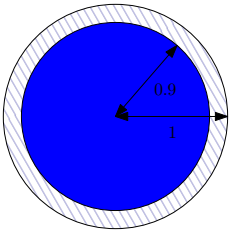

Figure 1: A 2-dimensional unit ball (blue) where the volume located at most 0.1 units from the surface is depicted in dashed lines.

(a) Roughly 1%

(b) Roughly 10%

(c) Roughly 20%

(d) Roughly 60%

Exercise (20 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 Motion Planning: Search and Sampling (80 points)

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

3.1 Code Overview

The starter code is written in Python and depends on NumPy and Matplotlib. This section gives a brief overview.

- `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.txt` - Maps that you will work with. The numbers denote the collision status of the cell (1 - collision, 0 - collision-free).
- `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 data structures that can be useful for your implementation of RRT and RRT*.

To run A* on map 2, you would run

```
$ python run.py -m map2.txt -p astar -s 148 321 -g 202 106
```

3.2 Environment Modelling

The planning consists of a 2D map. You have been provided with two maps `map1.txt` and `map2.txt`. You can use the former to *test* your implementation but report all results on the latter with start and goal to be $(321, 148)$ and $(106, 202)$. Note that environement-specific functions need to be filled in by you in `MapEnvironment.py` file, these functions can include distance computations or sampling and nearest neighbor retrieval. Once you complete implementing the environment-specific functions, you are ready to implement the planners.

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. Nevertheless, while implementing your collision detector, remember that the map is a pythonic array - and theretofore follows opposite convention: $< m, n >$ (i.e., <vertical, horizontal>).

3.3 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` file.

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.

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 it's correspond cell and test of that cell is obstacle-free or not.

Stated differently, each grid cell is either completely free or completely occupied by an obstacle.

3.4 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.