Homework 3

Robot Autonomy
CMU 16-662, Spring 2018
Due Date: 29th March, 11:59 pm

TAs: Anirudh Vemula, Fahad Islam

1 Introduction

In this homework you will be exploring the use of discrete planners for two different configuration spaces. You will implement a Breadth-First Search, a Depth-First Search and an A-Star Search. You will first run the planners in a simple two dimensional configuration space. Then you will move to the higher dimensional space of the WAM arm on the HERB robot.

2 Conversion from Continuous to Discrete Space

In order to implement these planners we will need a mapping from continuous configuration space to discrete space. In this part of the homework you will implement this mapping.

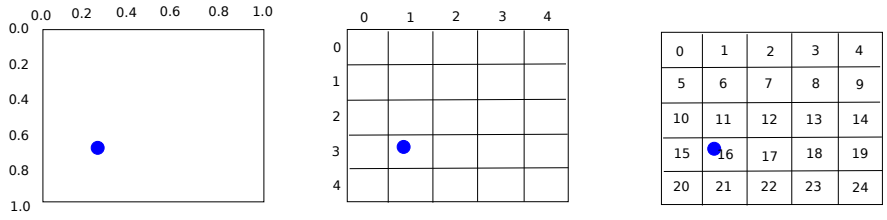

Figure 1: The mapping of a single point (shown in blue) from continuous configuration to discrete configuration to node id.

A single point in the world can be represented in three ways:

1. Continuous configuration - We can represent the configuration by a point in continuous space. (example: [0.26, 0.64])
2. Grid coordinate - We can lay a discrete grid down over the continuous space. Then every point can be assigned to a particular grid coordinate. (example: [1, 3])
3. Node id - We can assign each grid coordinate a unique number to identify the cell. (example: 16)

Figure 1 shows the three representations for a particular space. For this example figure, we discretize the world into 0.2m by 0.2m cells.

The file **DiscreteEnvironment.py** implements this mapping. Your job is to fill in the code for the following functions (all in terms of the resolution parameter):

1. *ConfigurationToGridCoord* - This function maps a configuration in continuous space to an associated grid coordinate in a discrete map. (i.e. [0.26, 0.64] $\rightarrow$ [1, 3])

2. *GridCoordToConfiguration* - This function maps a grid coordinate to a configuration in continuous space. For a particular grid space, this function should return the continuous coordinate that maps to the middle of the grid coordinate. (i.e. [1, 3] $\rightarrow$ [0.3, 0.7])

3. *GridCoordToNodeId* - This function maps a grid coordinate to a single number that will serve as the node id (example: [1, 3] $\rightarrow$ 16)

4. *NodeIdToGridCoord* - This function maps a node id to the associated grid coordinate (example: 16 $\rightarrow$ [1,3])

5. *ConfigurationToNodeId* - This function maps a configuration in continuous space to a single number that will serve as the node id (example: [0.26, 0.64] $\rightarrow$ 16)

6. *NodeIdToConfiguration* - This function maps a node id to a configuration in continuous space (example: 16 $\rightarrow$ [0.3, 0.7])

You will be using these functions for planning in both the 2-dimensional and 7-dimensional space, so be sure your functions work regardless of the dimension of the configuration space.

3 Planning for the Mobile Base

First, you will implement the planners and use them to move the robot base. As for the previous homework, we will consider an omni-directional robot and ignore orientation. So any given pose of the robot $q = [x, y] \in \mathbb{R}^2$.

Included in the assignment are stubs for each piece of code that you will need to implement. You should see a file **run.py**. This runs the code. The following will show you the options available:

```
python run.py --help
```

Running the following command will generate and execute the default plan for the mobile base:

```
python run.py --robot simple
```

The default plan is simply a two point plan that contains the start and goal. As can be seen, the robot drives straight through the table in the environment. Your goal in this part of the assignment is to implement a planner to create a collision free trajectory for the robot.

3.1 Simple Environment

The file **SimpleEnvironment.py** implements the 2D configuration space that you will plan in. First you will implement the following three functions in this file:

1. *GetSuccessors* - This function returns a list of neighbors for the node represented by the given id. You can choose the definition of neighbors. One suggestion is to use a 4-connected world.

2. *ComputeDistance* - This function computes the distance between two nodes.

3. *ComputeHeuristicCost* - The function computes the heuristic cost between two nodes with the given ids.

3.2 Planners

In this part of the assignment you will implement a set of discrete planners. First you will implement a Breadth-First planner. To do this, implement the *Plan* function in the file **BreadthFirstPlanner.py**. Next you will implement a Depth-First planner. To do this, implement the *Plan* function in the file **DepthFirstPlanner.py**. Finally you will implement an A-Star planner. To do this, implement the *Plan* function in the file **AStarPlanner.py**.

As in the last homework, we have provide a visualization tool for you to visualize the growth of the graph built by the planner for debugging. To use the visualizer, call the function *PlotEdge* on the planning environment every time you expand a node. Then start the planner with the visualize option:

```
python run.py --robot simple --planner dfs --visualize
```

4 Planning for a manipulator

Once you have a working version of the planning algorithms for the two dimensional robot, we will now use those planners to plan for a 7 degree-of-freedom manipulator. Running the following command will generate and execute the default plan for the 7 DOF arm:

```
python run.py --robot herb
```

Again, the default plan is simply a two point plan that contains the start and goal. As can be seen, the robot arm plans straight through the table in the environment. Your goal in this part of the assignment is to implement a planner to create a collision free trajectory for the arm.

4.1 Herb Environment

The file **HerbEnvironment.py** implements 7DOF WAM arm configuration space. Implement the following three functions in this file:

1. *GetSuccessors* - This function returns a list of neighbors for the node represented by the given id. You can choose how to define a neighbor. One recommendation is to change only one joint at a time.
2. *ComputeDistance* - This function computes the distance between two nodes.
3. *ComputeHeuristicCost* - The function computes the heuristic cost between two nodes with the given ids.

Note: Generating these plans will take a good amount longer than for any of your previous planners.

5 Deliverables and Grading

Please turn in a zip file (named `hw1-group<ID>.zip`) **on Gradescope** containing your code, a PDF writeup and the videos mentioned below. Only one person per group needs to submit but please make sure everyone's name and andrewid is on the pdf (**and add them as collaborators on Gradescope**). The following shows the point breakdown:

1. Run the Breadth-First search, Depth-First search and A-Star planner for the 2D configuration space. Run each planner 3 times, using resolutions of 0.05, 0.1 and 0.25. Report path length, plan time and number of nodes expanded for each planner and each resolution (9 runs total). Submit a video of one of the AStar runs. Include in your writeup a plot for the paths generated by each of the methods for resolution 0.1. (10 pts)

2. Run the AStar planner for the WAM arm. Use a resolution of 0.1. Report path length, plan time and number of nodes expanded during the search in the tree. Submit a video of the run. (10 pts)

3. For the 2D space, run the RRT algorithm with path shortening you implemented during the last homework using the start and goal defined for the 2D space in this homework. Run the RRT planner several times. Compare the path length of the AStar generated paths (for all three resolutions) to the paths generated by your RRT algorithm with path shortening. Do the same for the WAM arm. Comment on the strengths and weakness of using AStar versus RRT with path shortening. (5 pts)

4. EXTRA CREDIT: Implement the hRRT algorithm. To do this, implement the *Plan* method in **HeuristicRRTPlanner.py**. Copy over the relevant functions for **HerbEnvironment** and **SimpleEnvironment** from your previous homework. Use the same cost and heuristic that you implement for the AStar algorithm. Run the planner several times on both environments. Compare the plan times and path lengths to those obtained using the AStar algorithm. (+5 pts)

As a final note, the functions given in **SimpleEnvironment.py** and **HerbEnvironment.py** are simply a suggestion to help guide your implementation. You can change these functions in any way you see helpful (example: different inputs and outputs). We will test your code through the call to the *Plan* method on each planners, so please leave that interface intact.