Robot Motion Planning

Barry Yang
CS 76: Artificial Intelligence

November 18, 2017

To Run

Navigate to unzipped folder. Run `python3 prm.py` on terminal for the robot arm motion problem. Run `python3 rrt.py` for the car-like mobile robot problem. The important parameters for each problem are listed at the top of the files `prm.py` and `rrt.py`. These parameters can be changed. Please keep in mind that starting configurations must be passed in as tuples.

The `rrt.py` file automatically draws the solution. However, the `prm.py` does not automatically produce a visual representation of the solution. In order to run graphics, run `python3 graphics.py` on the command line. Unfortunately, there is no implementation for animation, so configurations for the solution must be hard coded into the top of `graphics.py` (example shown in file).

Introduction

There are two parts to this assignment: (1) robot arm and (2) robot motion planning. Robot arm involves build a probabilistic roadmap (PRM) will be used to find a collision-free path of configurations from a start configuration to an end configuration. Motion planning builds a rapidly exploring random tree to find a path of collision-free configurations from a start and an end goal.

Robot Arm

The robot arm is represented by a tuple of angles. The robot can be represented as a 2R, 3R, and 4R robot. A 2R robot is composed of two links and is configured by 2 angles to determine the orientation of the links. Analogously, a 3R robot is composed of three links and is configured by 3 angles to determine the orientation of the links (and 4R similar).

Structure

The robot arm problem is divided into three classes: `robot`, `environment`, and `prm`. The robot class contains methods and objects relevant to the robot arm itself. These include updating the configuration given a set of degree values. The environment class handles the robot as well as initiates a set of obstacles,

in which the PRM will be built around. The PRM class deals with the algorithm of finding a path of configurations for the robot arm. Implementation for these methods are discussed below.

Obstacle and Collision Detection

There are some obstacles in the environment that the robot can potentially collide into. Thus, the robot must be able to detect whether its arm can pass through an obstacle. The `environment` class sets up the robot as well as obstacles and the `robot` class sets up the robot with a starting and internally stored end position.

Collisions are detected via the `Shapely` library. The robot links are represented by line segments of a fixed length, and their configurations are determined by the the angle. For example, for a 2R robot that has angles 0 and 90, the first arm will extend eastward and the second arm will point north—forming an flipped "L" by the vertical axis. Thus, the x and y values of the links are determined by the angles in the configuration. In contrast, obstacles are represented by other shapes defined by the `Shapely` class. The collision detector makes use of the `intersection` function in the shapely library. If the two shape objects intersection, then there is a collision.

Angular Distance

Angular distance was calculated using radians. If a distance in one direction is d, the distance in the other direction is $2\pi - d$. The minimum was determined between the two values to determine the angular distance. To represent direction, a tuple was returned along with the distance to indicate the direction of movement–forward corresponds to `True` and backwards corresponds to `False`.

PRM

A Probabilistic Roadmap planner was implemented. The roadmap is build by calling `build_roadmap` in the PRM class. `build_roadmap` makes use of the `add_vertex` function in order to add nodes to the roadmap. The roadmap is a dictionary of a dictionary of edges. Each entry in the primary dictionary corresponds to a vertex in the map. This key maps to a dictionary that holds all the "edges," with the key being the neighboring vertex and the entry being a neighbor. Thus, this was constructed so that a vertex can have multiple edges stemming from the vertex. The code was implemented based on the pseudocode provided in `http://planning.cs.uiuc.edu/`.

`build_roadmap` begins by adding the start and goal vertices to the roadmap. Then, while the number of vertices is less than a predetermined number, generate a random valid configuration and keep adding those configurations to the roadmap via `add_vertex`.

`add_vertex` adds a vertex to the roadmap. If the configuration is not already in the graph, then the nearest k neighbors are called and a new entry for the vertex is put into the roadmap. For each neighbor in the list of neighbors,

if there is no collision, then an edge is connected between the vertex and the neighbor. An edge is simply an entry into the dictionary corresponding to a vertex. Each key in this dictionary is a neighboring vertex mapping to the length between the vertices.

Query Testing

BFS was used to traverse the graph created by PRM from the start to the goal state.

Car-like Mobile Robot

A configuration for the car-like robot is represented by a tuple of three elements: an x and a y value representing coordinates as well as a θ value representing the angle of the car. This tuple can interact with two other elements to find a successor configuration: which is a list of "controls" as well as a duration (how long the robot will travel given the control). The control represents a frontwards and backwards motion as well as whether the robot moves clockwise or counter-clockwise. There are six possible controls, as described on the assignment page. The semantics of how the successor states are computed given a configuration are not important for the problem, but are important for understanding how the robot motion works.

The entirety of the mobile robot problem is contained in `rrt.py` In my implementation of the problem, the car is represented as a `Shapely` point and obstacles are represented by polygons.

Building and Traversing the Graph

The RRT is formed from a seed starting node. In each iteration, a random configuration is generated and the nearest neighbor in the tree from the random configuration is found. From this neighbor, the node expands in six directions until either the goal configuration has been found, or the node cannot expand further. The expanded node is then added to the tree to increase the list of neighbors for the next iteration.

Similar to PRM, my RRT implementation also calls upon a `build_tree` method. Unlike the PRM data structure, the graph is represented by a list, which holds a `SearchNode` object containing the configuration, the parent configuration node, as well as the control needed to get to the stored configuration. Thus, nodes only have one successor. The tree was implemented so that backtracking between nodes in order to find a solution would be easy.

`build_tree` begins by inserting the starting configuration as a "seed," or as the first node in the RRT. While the length of the graph is less than the limit, a random configuration is generated. A random configuration is generated by choosing x and y values that are contained within the window. Then, that random value is fed into the `get_nearest_neighbor` function to return the nearest node in the graph relative to the randomly generated configuration (thereafter,

the randomly generated configuration is no longer used). To determine nearest distance, the `distance` function from the `shapely` library is used to calculate the distance between two xy coordinates.

Once the nearest neighbor is found, the function `no_collisions_trajectory` is run to find the last valid configuration given a control before a collision is reached. To check for collisions, the `Shapely` library is again used to check for interactions between two `Shapely` objects. However, because the car is represented as a point, we check whether any obstacles contain the point. If an obstacle contains the point, then a collision has been met, so a move is not valid. Once the last valid configuration is found, it is returned and added to the tree. In addition, if the configuration is near the goal (given a threshold) then a solution is considered to be found and the algorithm backtracks in order to find a path from start to goal.

Results

To see results, please see appended.

At 100 vertices and 20 timesteps between vertices, PRM returns below. The first and the last elements are the starting and end configurations respectively.

SOLUTION:

[(3.141592653589793, 4.71238898038469, 1.570796326794896, 4.71238898038469),

(3.238762669143777, 0.780248312425674, 1.6863333594653087, 4.590840931466817),

(4.485570585480902, 0.6805452727332207, 4.695385895866597, 0.4312057568122185 7),

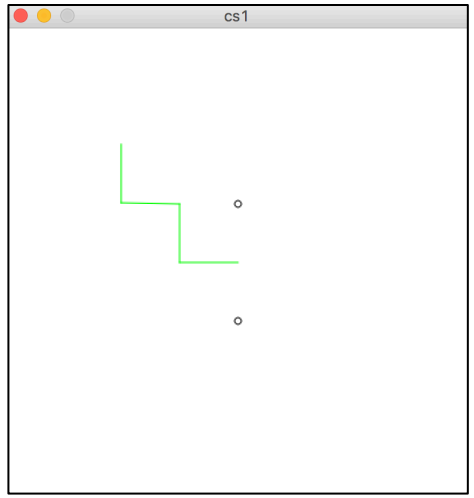
(1.168125383616999, 0.4721068358472472, 4.595115650517950 5, 0.01315944977792538 7),

(0, 1.570796326794896, 4.71238898038469, 1.570796326794896)]

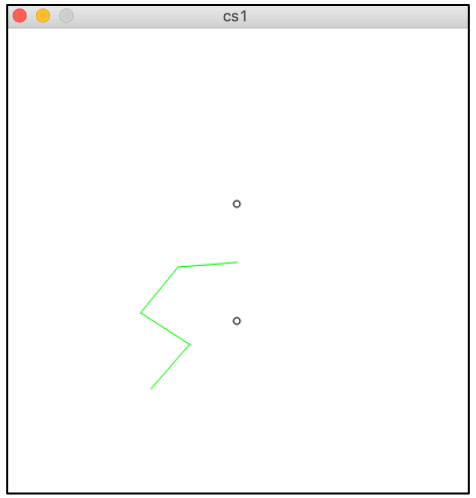
To the right is a graphical representation of the solution. Though it looks like there are dramatic jumps, the PRM solution was built with a relatively small amount of 100 vertices with 20 iterations of timesteps. Thus, there may be large gaps in movements.

(run in ~20 seconds).

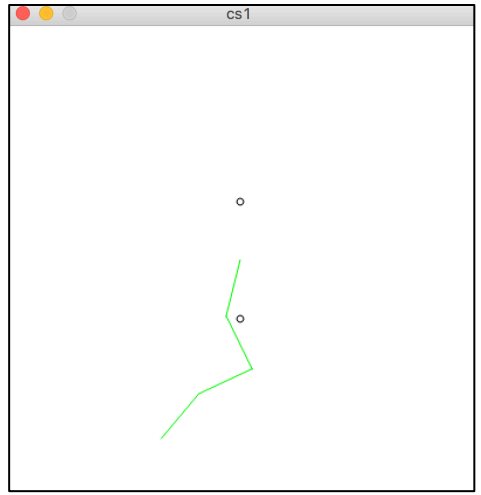
1

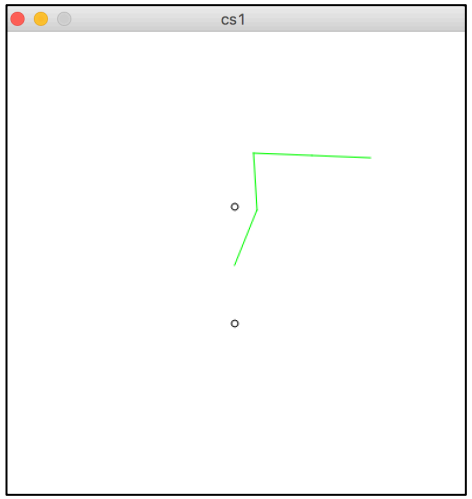


2

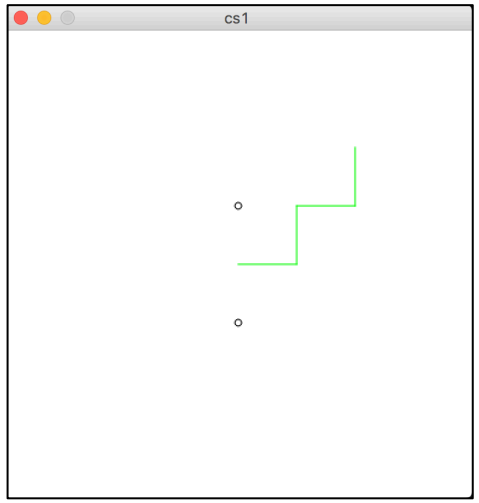


3





4



5

This is the progression of RRT. The black represents the tree and the green line represents the path to the goal state. Red lines introduce obstacles. The follow represents the graphs as more obstacles are placed.