Robot Motion Planning

Ping-Jung Liu November 15th 2017

1. Introduction

In this project, I implemented the Probabilistic Roadmap (PRM) and Rapidly Exploring Random Tree (RRT) to solve two types of robot motion planning problems.

2. Probabilistic Roadmap (PRM)

The objective of this problem is to move a robotic arm from one configuration to the goal configuration without colliding with the obstacles in the environment. The main idea of PRM is to randomly generate a number of configurations in the environment and connect each of them with its k nearest neighbors; this is the roadmap of the problem. After the roadmap is built, use A-star search to find a path from the start configuration to the goal configuration.

I separated this problem into two components, prm-map and prm-problem. The previous generates the roadmap given the environment parameters and the latter find the solution path between start and goal configurations. First I will discuss the features of prm_map.

The robotic arm is represented with a list of theta, which shows the angles between each arm. I used trigonometry to calculate the positions of each arm with the angles.

```
# calculate the position of robot arms
def calc_pos(self, base, length, theta):

    x = [base[0]]
    y = [base[1]]

    for i in range(0, len(theta)):

        new_x = 0
        new_y = 0
        for j in range(0, i + 1):

            coss = cos(sum(theta[0:j + 1]))
            sinn = sin(sum(theta[0:j + 1]))
            if abs(coss) < 0.0001:
                coss = round(coss)
            if abs(sinn) < 0.0001:
                sinn = round(sinn)

            new_x = new_x + length[j] * coss
            new_y = new_y + length[j] * sinn


        x.append(new_x)
        y.append(new_y)

    pos = []
    for i in range(0, len(x)):
        pos.append((x[i], y[i]))
    return pos
```

The robotic arm cannot collide with obstacles; I used the intersect method from shapely library to check if the arms collide with any of the obstacle.

```
# check if a robot arm is collision free
def collision(self, pos):

    polys = []
    for i in range(0, len(self.obstacles)):
        polys.append(Polygon(self.obstacles[i]))

    robot = LineString(pos)
```

```
    for i in range(0, len(polys)):
        if polys[i].intersects(robot):
            return True

    for i in range(0, len(pos)):
        if self.out_of_bound(pos[i]):
            return True

    return False
```

Note: pos is the positions of ends of each arm

After I made sure the robot kinematics and collision checking were solid, I started to build the actual roadmap. As stated, the program will plant many random configurations and connect the k nearest neighbors. In this project I chose k to be 15, suggested by Planning Algorithms, Steven M. LaValle, 2006,Cambridge University Press. And by shortest, I meant the least anglular difference between two configurations. As for generate size, 1000 random nodes could solve the default setting, which will be shown later, with decent results.

Code of roadmap as follow:

```
def roadmap(self):
    #initialize roadmap
    graph = {}
    i = 0
    # grow node until generate_size reached
    while i < self.generate_size:
        # to observe the progress
        print(i)
        # find a random collision free configuration
        config = self.collisionFree_config(graph)
        config = tuple(config)
        # ignore if exist in graph
        if config in graph:
            continue
        # add the configuration to roadmap
        else:
            self.add_config(config, graph)
            i = i + 1

    return graph
```

collisionFree_config find a random collision free configuration in the environment. Code as follow:

```
# find a random collision free configuration
def collisionFree_config(self, graph):

    flag = False
    # loop until a legal configuration found
    while not flag:

        # find a random configuration
        test_config = []
        for i in range(0, self.arm_n):
            rand_theta = random.uniform(0, 2 * pi)
            test_config.append(roundpi(rand_theta))
        # calculate the positions of robot arms
        test_pos = self.calc_pos((0, 0), self.arm_l, test_config)
        # if legal, exit the loop and return the configuration
        if not self.collision(test_pos) and not tuple(test_config) in graph:
            flag = True

    return test_config
```

After a random collision free configuration was found, add it to the roadmap with add_config. Code as follow:

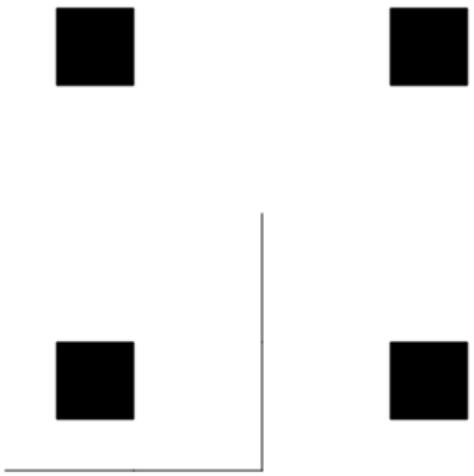
```
# add a configuration to the roadmap
def add_config(self, config, graph):

    # find the neighbors of config
    neighbors = self.nnr(config, graph)
    # create edges between config and its neighbors
    graph[config] = neighbors
    for neighbor in neighbors:
        if not config in graph[tuple(neighbor)]:
            graph[tuple(neighbor)].append(config)
```

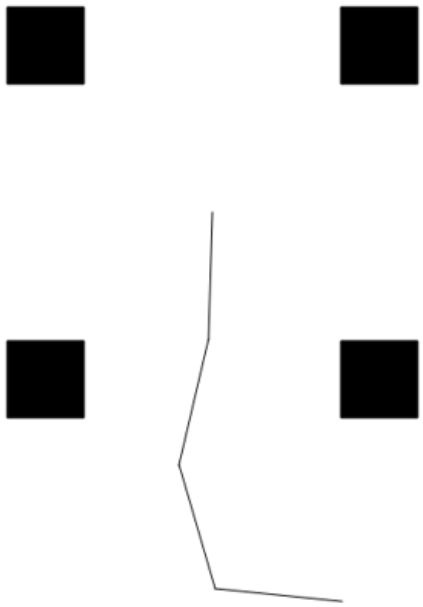
I will not dig into the details of finding neighbors here, but will discuss an important point. When we found a neighbor that is collision free, that does not indicate the trajectory to this neighbor is also collision free, so I divided the trajectory into several, indicates by "resolution" in the code, sub-configurations and make sure they are all collision free. This step can greatly decrease the chance of having obstacles on the trajectory.

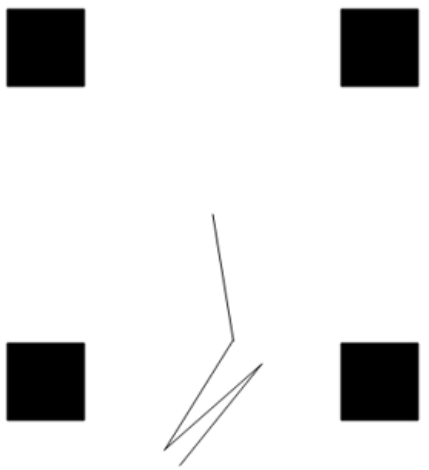
After the roadmap is successfully built, we can then use prm-problem.py to initialize a new problem and solve it. First we need to add the start and goal configuration into the roadmap. In order to use A star search, I still defined several functions like goal*test, get*successors, heuristics.

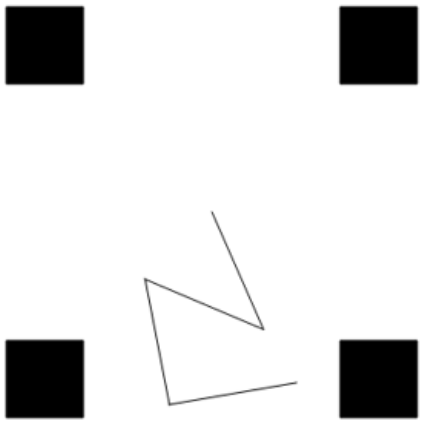
```
# check for goal
def goal_test(self, state):
    return tuple(state) == tuple(self.goal)

# get the successors of a state
def get_successors(self, state):

    neighbors = self.mapp.roadmapp[tuple(state)]
    successors = []
    for i in range(0, len(neighbors)):
        successors.append((neighbors[i], 1))
    return successors

# this heuristics calculate the sum of angle differences between state and goal
def heuristics(self, state):

    summ = 0
    for i in range(0, len(state)):
        diff = abs(state[i] - self.goal[i])
        if diff > pi:
            diff = 2 * pi - diff
        summ = summ + diff

    return summ
```

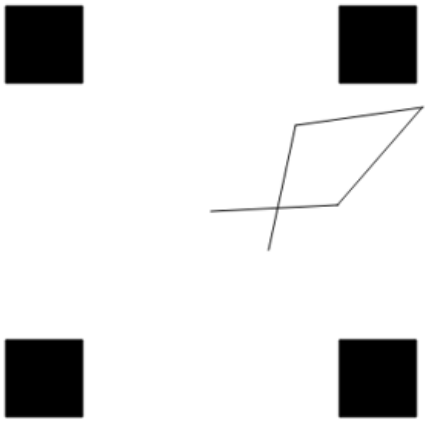
I tested the PRM motion planner with the standard robotic arms scenario. As follow:

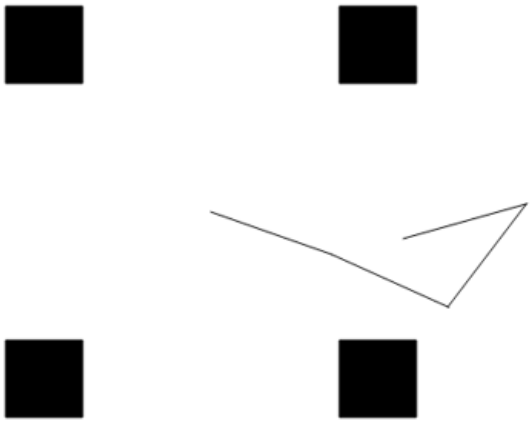


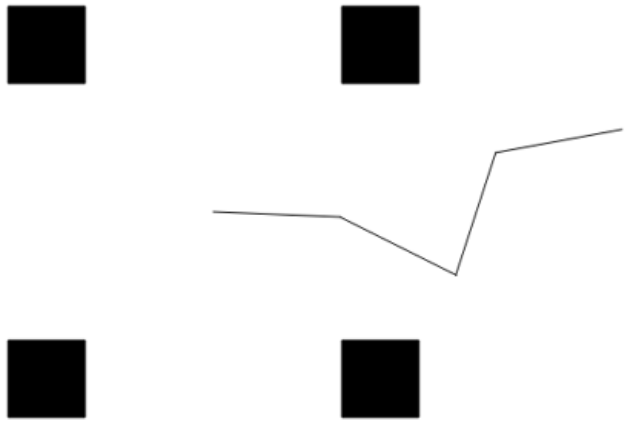


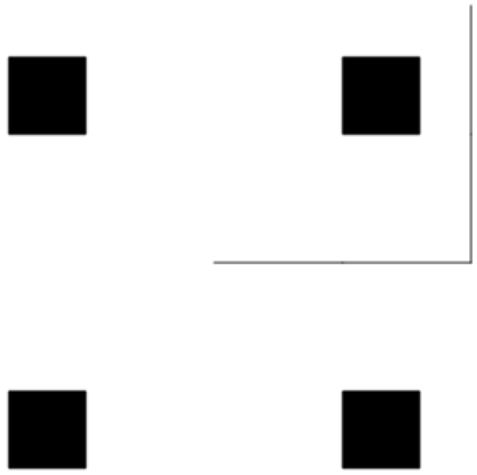












The whole process took about 1 minute and 45 seconds with 1000 generated configurations. We can see that the robotic arm indeed squeezed through the obstacles to reach the goal.

3. Rapidly Exploring Rando Tree (RRT)

The objective the problem is to find a way to drive a robotic car from the start state to the goal state without colliding with obstacles in the environment. The main idea of RRT is to grow a tree from the start position by expanding random leaves, until the tree get close enough to the goal position. The program will then backtrack from the goal and return the path from start to goal.

The car has six possible moves, forward, backward, forward right, forward left, backward right, and backward left. Everytime the program expand a leaf, it will simulate all six possible moves with a given timestep and check if the results are collision free. That said, we can start generating the tree. If a move is indeed legal, the resulting position will be saved into a dictionary, and the value would be a tuple of two elements, first is its parent's position, second is the move the parent took to get here. That said, we can start growing the tree with the function grow-map.

```
# grow the whole map
def grow_map(self):

    # potential goal?
    potential = self.expand(self.start)

    # loop until potential goal is close to actual goal
    i = 0
    while self.flag == 0:
        # observe the progess
        print(i)
        # random point
        random_x = random.uniform(0, self.width)
        random_y = random.uniform(0, self.height)
        # find the closest leaf
        neighbor = self.closest_vertex(random_x, random_y)
```

```
        # expand the leaf
        potential = self.expand(neighbor)
        i = i + 1

    return potential
```

When a leaf is close enough to the goal, grow-map would return the potential goal and start backtracking for the solution path. The function closest vertex finds the closest leaf to a random point on the map, then expands it with self.expand.

closest_vertex code as follow:

```
# find the closest leaf (vertex)
def closest_vertex(self, x, y):

    neighbors = []
    for node in self.leaves:
        neighbors.append((self.dist((x, y), (node[0], node[1])), node))
    neighbors.sort()

    return neighbors[0][1]
```

expand code as follow:

```
# expand the leaf
def expand(self, vertex):

    # since we are expanding the leaf, put it in parent
    # then remove from leaves
    self.parent.append(vertex)
    if vertex in self.leaves:
        self.leaves.remove(vertex)
    # get the transform of the leaf
    start_t = transform_from_config(vertex)

    # try all six control moves
    for i in range(0, 6):
        test_transform = single_action(start_t, controls_rs[i], self.delta)
        test_config = config_from_transform(test_transform)

        # test for collision and map bounds
        if self.good_control(vertex, controls_rs[i]) and not test_config in self.graph:
            self.graph[test_config] = (vertex, controls_rs[i])
            self.leaves.append(test_config)

            # if the resulting configuration is close to goal, return
            if self.close_to_goal(test_config):
                self.flag = 1
                return test_config
    return None
```

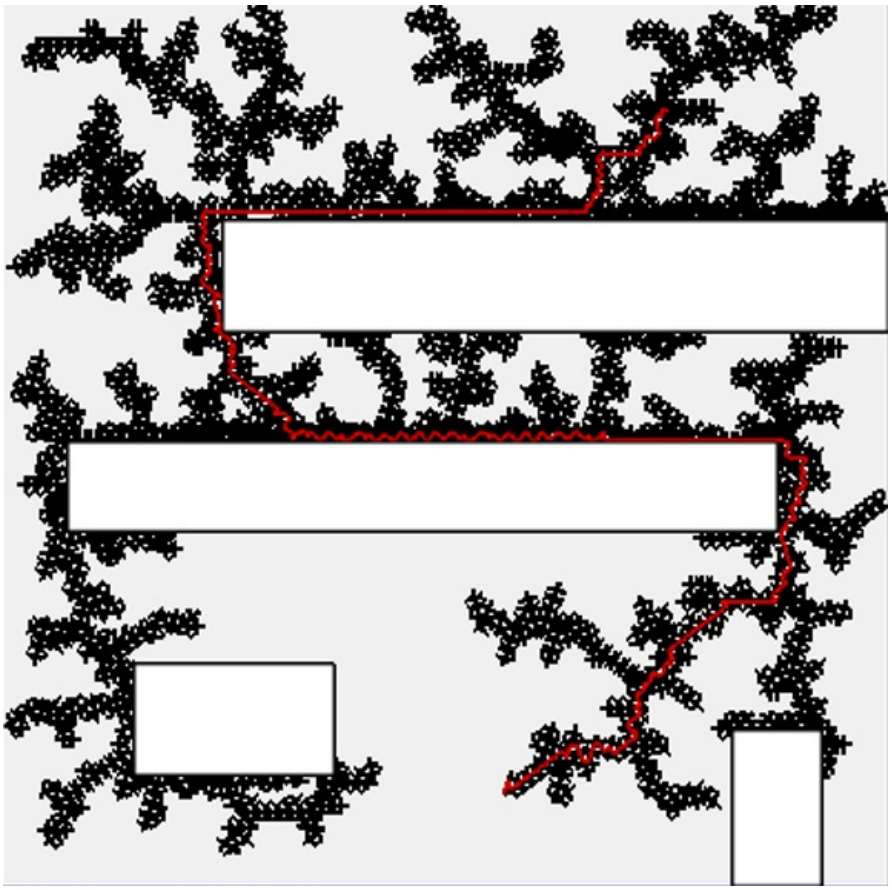
This is the most important function of RRT so let's go through it carefully. First add the vertex into a list of parent because it is becoming a parent. Then delete it from a list of leaves because of the same reason. The for loop simulates all 6 possible moves of the robot. The function good_control makes sure the move does not collide with obstacles. The collision detection process is similar to that in PRM; the program will sample several position in between the starting position and the resulting position of the move and chekc if any of them is contained in obstacles. After that, if the resulting position is close to the goal, set flag to 1 so the program knows the goal has been found.

After the goal is found, the program will backtrack from the goal using the dictionary to the start state.

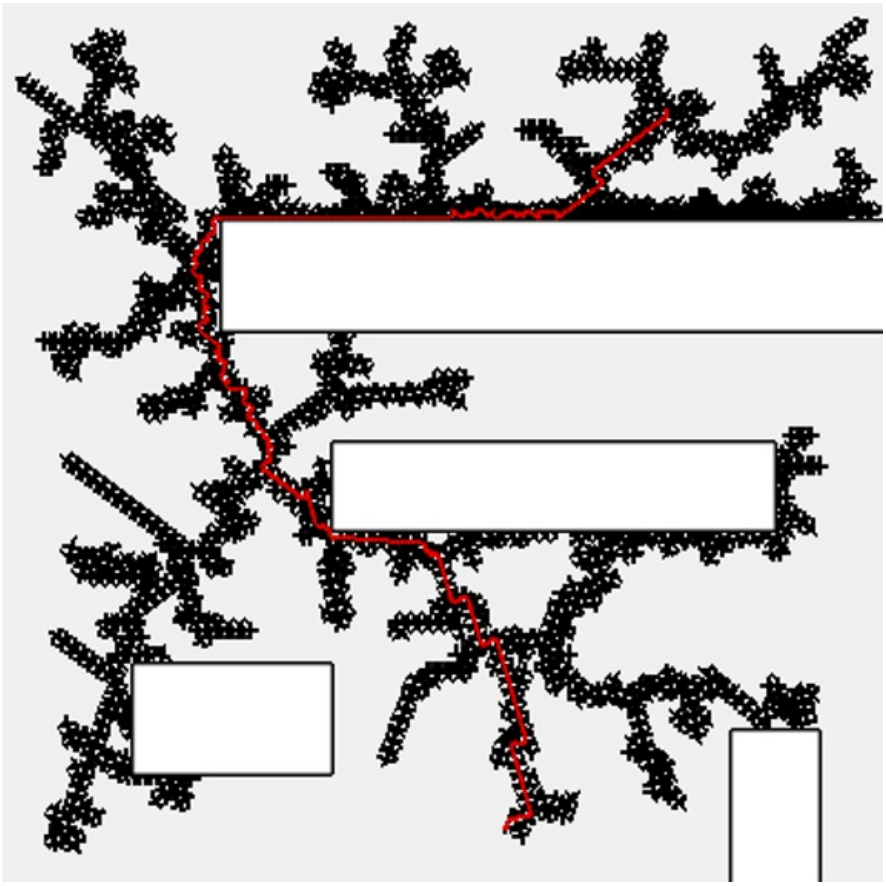
```
def solve(self):

    # grow the map until reaches the proximity of the goal
    goal_close = self.grow_map()
    print(goal_close)

    # backtrack to obtain the solution path
```

```
    path = []
    cur = goal_close
    while not cur == self.start:

        path.append(self.graph[cur])
        cur = self.graph[cur][0]

    return path
```

I played around with the program an chose delta, the time step, to be 5 and angular velocity to be 0.5. The program could solve a 400 * 400 environment around one minute. Two test cases as follow:



This one took 1:00


This one took 0:30

We can observe that the tree rapidly search the whole environment and evade the obstacles. The red trajectory indicates the solution path.

4. Literature Review (Undergrad, Extra Credit)

For extra credit, I reviewed "A Comparative Study of Probabilistic Roadmap Planners by Roland Geraerts and Mark H. Overmars Institute of Information and Computing Sciences, Utrecht University".

This research paper compared the performances of different extentions of the traditional probabilistic roadmap in collision checking, sampling, and node adding. While implementing the project, I was skeptical about the randomness of PRM and its capability to find decent restult, so I found this paper interesting because it provides many potential methods to increase the soundness and functions of PRM. I am particularly interested in the sampling methods, because the one I implemented was a completely random one and even though the result was decent, I felt bad about it. The paper studied the following techniques:

Random, in which the program randomly select collision free configurations, which is also the one I implemented.

Grid, in which the program choose samples on a grid. Because we might not know the grid resolution before hand, we can start with a coarse grid and refine it in the process.

Halton, a deterministic sampling method used in discrepancy theory.

Cell-based, in which the program progressively split the map into smaller cells. Each time the program will randomly choose configurations in all of the existing cellsto make sure the sample space potently cover the whole region.

Obstacle based, in which the program randomly select a configuration; if it collides with obstacles, randomly select a direction and modify the configuration until it is collision free.

I found cell-based and obstacle based especially interesting. Because they seemed intuitive to implement but yet provide so much more confidence in the sample coverage of the map.

There are numerous other methods related to collision checking and node adding discussed in the paper, but I will not discuss them here. Over all the paper provides a detailed introduction on the concepts of probabilistic roadmap, then discussed many interesting extentions, including comparason between them, to boost the performance of PRM.