Anujay Sharma

Standard Search Algorithms

Q1) For PRM, what are the advantages and disadvantages of the four sampling methods in comparison to each other?

A1.

Uniform Sampling	Random Sampling	Gaussian Sampling	Bridge Sampling
Uniform sample across C-space	Random Sample Across C-space	Samples only along the edge	Samples only along narrow paths
Will get more nodes to find the path throughout the entire free C-space	Has mixed result depending on the sampling	Will not cover the free space very well	Will not cover free space very well
Simple to implement	Simple to Implement	Must place checking conditions throughout the C-Space	Must place checking conditions throughout the C-Space
Usually fails to go through narrow paths	Has mixed results depending on sampling	Can define the obstacles avoidance strategies	Can find paths through narrow passages
Used when more free space is there	Used to test results	Used with less free space	Used with very less free space

Q2) For RRT, what is the main difference between RRT and RRT*? What change does it make in terms of the efficiency of the algorithms and optimality of the search result?

A2.

RRT* is an optimized version of the RRT algorithm. When the number of nodes approach infinity the RRT* algorithm will give out the shortest possible path to reach the goal. The overall algorithm is the same as RRT however it employs three changes:

a) Recording distance relative from parent node
b) Figuring out the neighbors of the current node
c) Rewiring – Rearranging the family based on costs (cheapest neighbor)

However, all this comes at a price of computational efficiency. The time to find a path with RRT* could be eight times more as compared to RRT. This is because a lot of memory is consumed in examining, rewiring and collision checking.

The search path for RRT* as much more optimal as compared to RRT which would break as soon as a path is found. RRT* keeps running until it runs out of sampling points, meanwhile optimizes it path at every iteration for every new sample which could be found for reducing the overall cost of traversal.

Q3) Comparing between PRM and RRT, what are the advantages and disadvantages?

A2.

PRM's are multi-query planners. This means that we sample the C-space once and then make connections between the start and the goal by connecting them to the roadmap. These roadmaps can be re-used to find paths for different configurations of the start and goal positions.

RRT is a single-query algorithm. This means that every time we need to find a path between start and goal, we would construct it online. These trees are re-executed to generate paths for a new set of start and goal position.

RRT terminates as soon as the goal is found. This mean that the runtime is shorter as compared to other algorithms. However, the path found may not be the optimal one. This could also be used in dynamic environment as a new path can be constructed as soon as any change is noticed.

The PRM may take time initially to construct the node map. It then uses a graph-search method like A* or Dijkstra to traverse though the roadmap. However, there could be issue when you need to plan in real time with dynamic obstacles, as new search from the current to the goal position would be needed again. PRM could also sample points inside closed or narrow spaces if nodes located inside the obstacles are not discarded. PRM consider the idea that the environment won't change over time.

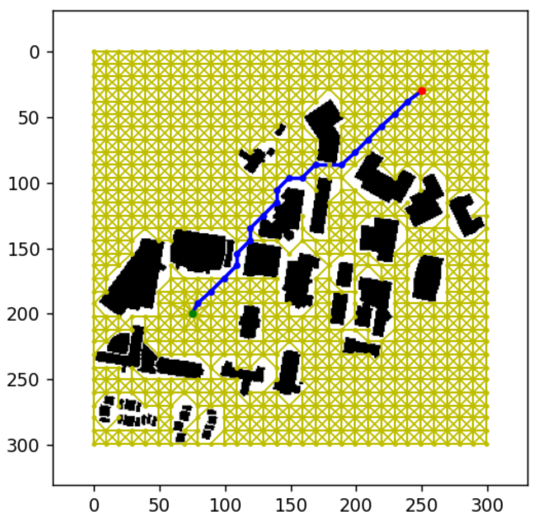
PRM retains the graph structure and reuses it every time whereas RRT can find path in dynamic environment but with the expense of optimality.

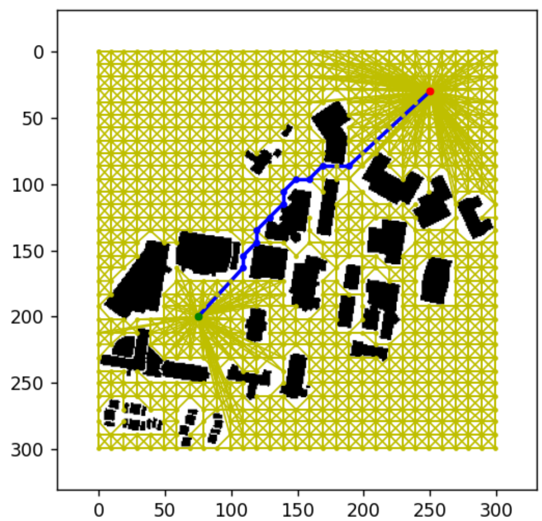
Algorithm Results and Explanation

PRM.

We get the following results for PRM for different sampling methods:

Uniform Sampling:



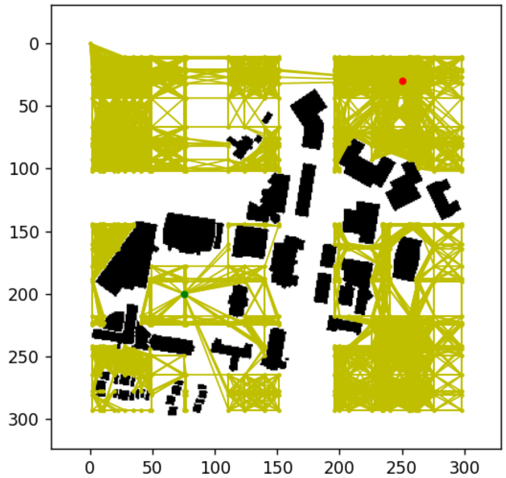


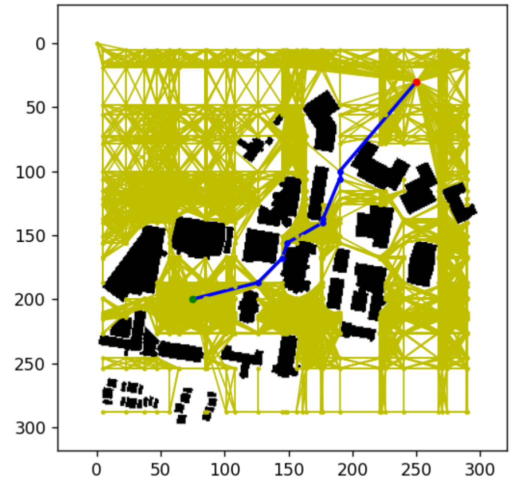
In the first figure we see that for uniform sampling we do not require large radii for the KDtree. A larger radius causes unwanted connection in free space.

Since this map has a much more free space as compared to narrow regions, a uniform sampling based method is easy to implement and a path can also be easily found out. The cell size is almost same to that of the narrow gaps.

Method: The entire space is equally distributed into equally spaces nodes for sampling

Random Sampling:



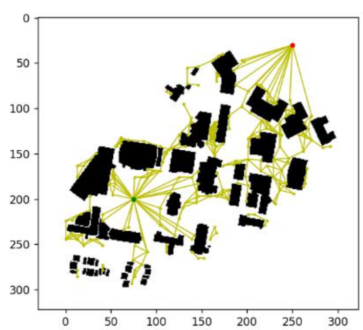


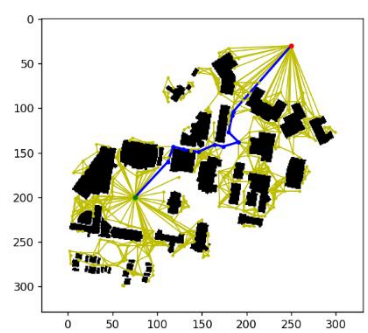
In these two images we see that random sampling fails in the first case whereas it is successful for the second case. This entirely depends on how the nodes are sampled on the map. If we are lucky then we might get a feasible path.

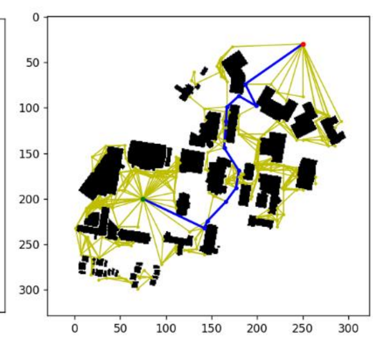
Also notice to get a feasible path we either increase the number of samples on the map or increase the radii of the KDTree.

<u>Method:</u> The entire space is randomly distributed for sampling

Gaussian Sampling:



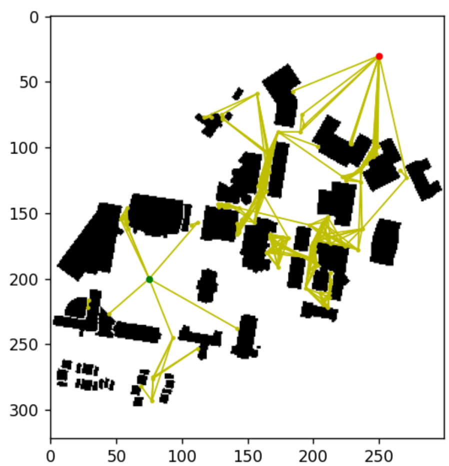


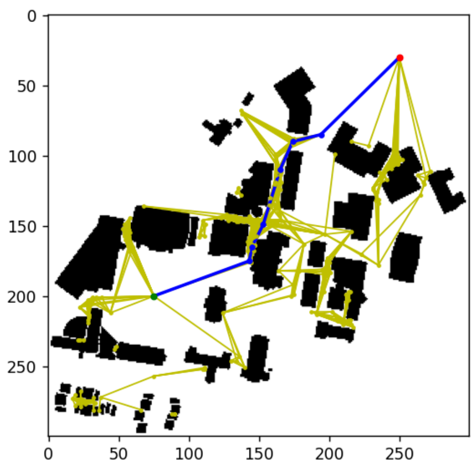


Here we see three images for Gaussian based sampling. In the first one we fail to find a path. This happens as the points are sampled near the edges of the obstacles. This leaves us with very few ways to make connections. We either increase the sampling size as seen in the second image or increase the radii of the KDtree as seen in the third image.

<u>Method:</u> Here first we take random samples across the space. Then we pick another point from a Gaussian distribution centered at our sampled point. If both the points are collision free or inside obstacle, they are discarded. Otherwise, the one which is free is kept and the other discarded.

Bridge Sampling:





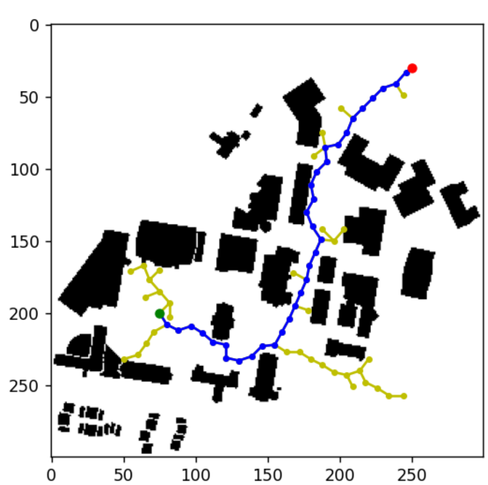
Like gaussian sampling in case of bridge sampling we sample points only across narrow passage. This is good for confided space but for spaces with of lot for free space, it becomes problematic sic ewe run out of samples to joins. In large free space we can only make connections by using larger radii for KDTree.

<u>Method:</u> Here first we take random samples across the space. Then we pick a point which is inside an obstacle. Then we pick another point from a Gaussian distribution centered at our sampled point. If both the points are inside obstacle, we will pick another point, i.e., the midpoint of these two points. If the midpoint is in free space, we will use it for sampling.

RRT.

We implemented two RRT based method:

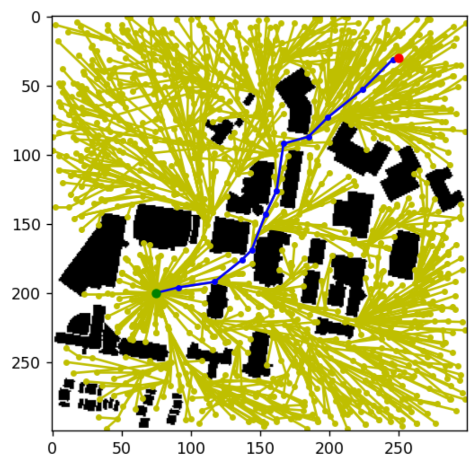
Standard RRT:



We see in the above image an implementation of standard RRT algorithm. This tree expands randomly in any direction until it reaches the goal. The path found is not optimal.

Method: Initiate a new random node and find its nearest node. Then explore in that direction if no collision is there. Keep expanding throughout the tree until the goal is achieved.

RRT*:



In the above image we can see an implementation of the RRT* algorithm. This algorithm works in a similar fashion like standard RRT but keeps optimizing the path even when the goal is found. If there are infinite sampling points RRT* will find the shortest path. We can see a more optimal path for RRT* as compared to RRT.

Method: Initiate the same way you would for RRT. However, when adding new nodes to the graph make sure that you rewire the costs for the cheapest way. Correspondingly also reduce the costs for the neighbors if possible.

Code:

PRM:

```
# Standard Algorithm Implementation
# Sampling-based Algorithms PRM

import matplotlib.pyplot as plt
import numpy as np
import networkx as nx
from scipy import spatial

# Class for PRM
class PRM:
    # Constructor
    def __init__(self, map_array):
        self.map_array = map_array            # map array, 1->free, 0-
>obstacle
        self.size_row = map_array.shape[0]    # map size
```

```
        self.size_col = map_array.shape[1]     # map size

        self.samples = []                      # list of sampled points
        self.graph = nx.Graph()                # constructed graph
        self.path = []                         # list of nodes of the found
path


    def check_collision(self, p1, p2):
        '''Check if the path between two points collide with obstacles
        arguments:
            p1 - point 1, [row, col]
            p2 - point 2, [row, col]

        return:
            True if there are obstacles between two points
        '''
        ### YOUR CODE HERE ###
        pt_set = set()
        points = np.linspace(p1, p2, dtype=int)
        [pt_set.add(tuple(x)) for x in points if tuple(x) not in pt_set]
        for j in pt_set:
            if self.map_array[j[0]][j[1]] == 0:
                return True
        return False

    def dis(self, point1, point2):
        '''Calculate the euclidean distance between two points
        arguments:
            p1 - point 1, [row, col]
            p2 - point 2, [row, col]

        return:
            euclidean distance between two points
        '''
        ### YOUR CODE HERE ###
        return np.sqrt((point1[1]-point2[1])**2 + (point1[0]-point2[0])**2)

    def distribution(self, n_pts, random):
        '''
        Function randomly/uniformly distributes points across the space
        '''
        w = self.size_row
        h = self.size_col
        n_y = int(np.sqrt(w * n_pts / h + ((w - h) ** 2) / 4 * (h ** 2)) -
((w - h) / 2 * h))
        n_x = int(n_pts / n_y)
        if not random:
            vec_x = np.linspace(0, w - 1, n_x, dtype=int)
            vec_y = np.linspace(0, h - 1, n_y, dtype=int)
            grid_x, grid_y = np.meshgrid(vec_x, vec_y)
            row_idx = grid_x.ravel()
            col_idx = grid_y.ravel()
            return zip(row_idx,col_idx)
        else:
            vec_x = np.random.randint(0, w - 1, n_x, dtype=int)
            vec_y = np.random.randint(0, h - 1, n_y, dtype=int)
```

```python
        grid_x, grid_y = np.meshgrid(vec_x, vec_y)
        row_idx = grid_x.ravel()
        col_idx = grid_y.ravel()
        return zip(row_idx, col_idx)

    def uniform_sample(self, n_pts):
        '''Use uniform sampling and store valid points
        arguments:
            n_pts - number of points try to sample,
                    not the number of final sampled points

        check collision and append valide points to self.samples
        as [(row1, col1), (row2, col2), (row3, col3) ...]
        '''
        # Initialize graph
        self.graph.clear()
        ### YOUR CODE HERE ###
        self.samples.append((0, 0))
        grid = self.distribution(n_pts, random=False)
        for g in grid:
            if self.map_array[g[0]][g[1]] == 1:
                self.samples.append(g)

    def random_sample(self, n_pts):
        '''Use random sampling and store valid points
        arguments:
            n_pts - number of points try to sample,
                    not the number of final sampled points

        check collision and append valide points to self.samples
        as [(row1, col1), (row2, col2), (row3, col3) ...]
        '''
        # Initialize graph
        self.graph.clear()
        ### YOUR CODE HERE ###
        self.samples.append((0, 0))
        grid = self.distribution(n_pts, random=True)
        for g in grid:
            if self.map_array[g[0]][g[1]] == 1:
                self.samples.append(g)

    def gaussian_sample(self, n_pts):
        '''Use gaussian sampling and store valid points
        arguments:
            n_pts - number of points try to sample,
                    not the number of final sampled points

        check collision and append valide points to self.samples
        as [(row1, col1), (row2, col2), (row3, col3) ...]
        '''
        # Initialize graph
        self.graph.clear()
        ### YOUR CODE HERE ###
        # self.samples.append((0, 0))
        grid = self.distribution(n_pts, random=True)
        for g in grid:
            if self.map_array[g[0]][g[1]] == 0:
```

```python
                # obstacle points
                # normal distribution
                f_x = int(np.random.normal(g[0], 5))
                f_y = int(np.random.normal(g[1], 5))
                if 0 < f_x < self.size_row and 0 < f_y < self.size_col:
                    if self.map_array[f_x, f_y] == 1:
                        self.samples.append([f_x, f_y])
            else:
                # free points
                # normal distribution
                ob_x = int(np.random.normal(g[0], 5))
                ob_y = int(np.random.normal(g[1], 5))
                if 0 < ob_x < self.size_row and 0 < ob_y < self.size_col:
                    if self.map_array[ob_x, ob_y] == 0:
                        self.samples.append(g)


    def bridge_sample(self, n_pts):
        '''Use bridge sampling and store valid points
        arguments:
            n_pts - number of points try to sample,
                    not the number of final sampled points

        check collision and append valide points to self.samples
        as [(row1, col1), (row2, col2), (row3, col3) ...]
        '''
        # Initialize graph
        self.graph.clear()

        ### YOUR CODE HERE ###
        grid = self.distribution(n_pts, random=True)
        for g in grid:
            # check obstacle
            if self.map_array[g[0]][g[1]] == 0:
                f_x = int(np.random.normal(g[0], 10))
                f_y = int(np.random.normal(g[1], 10))
                if 0 < f_x < self.size_row and 0 < f_y < self.size_col:
                    # check obstacle
                    if self.map_array[f_x, f_y] == 0:
                        # find middle point devoid of obstacle
                        mid_x = int((f_x + g[0])/2)
                        mid_y = int((f_y + g[1]) / 2)
                        if self.map_array[mid_x, mid_y] == 1:
                            self.samples.append([mid_x, mid_y])


    def draw_map(self):
        '''Visualization of the result
        '''
        # Create empty map
        fig, ax = plt.subplots()
        img = 255 * np.dstack((self.map_array, self.map_array,
self.map_array))
        ax.imshow(img)

        # Draw graph
        # get node position (swap coordinates)
```

```
        node_pos = np.array(self.samples)[:, [1, 0]]
        pos = dict( zip( range( len(self.samples) ), node_pos) )
        pos['start'] = (self.samples[-2][1], self.samples[-2][0])
        pos['goal'] = (self.samples[-1][1], self.samples[-1][0])

        # draw constructed graph
        nx.draw(self.graph, pos, node_size=3, node_color='y', edge_color='y'
,ax=ax)

        # If found a path
        if self.path:
            # add temporary start and goal edge to the path
            final_path_edge = list(zip(self.path[:-1], self.path[1:]))
            nx.draw_networkx_nodes(self.graph, pos=pos, nodelist=self.path,
node_size=8, node_color='b')
            nx.draw_networkx_edges(self.graph, pos=pos,
edgelist=final_path_edge, width=2, edge_color='b')

        # draw start and goal
        nx.draw_networkx_nodes(self.graph, pos=pos, nodelist=['start'],
node_size=12,  node_color='g')
        nx.draw_networkx_nodes(self.graph, pos=pos, nodelist=['goal'],
node_size=12,  node_color='r')

        # show image
        plt.axis('on')
        ax.tick_params(left=True, bottom=True, labelleft=True,
labelbottom=True)
        plt.show()

    def sample(self, n_pts=1000, sampling_method="uniform"):
        '''Construct a graph for PRM
        arguments:
            n_pts - number of points try to sample,
                    not the number of final sampled points
            sampling_method - name of the chosen sampling method

        Sample points, connect, and add nodes and edges to self.graph
        '''
        # Initialize before sampling
        self.samples = []
        self.graph.clear()
        self.path = []
        radius = 0
        # different radii for different methods
        # Sample methods
        if sampling_method == "uniform":
            self.uniform_sample(n_pts)
            radius = 20
        elif sampling_method == "random":
            self.random_sample(n_pts)
            radius = 35
        elif sampling_method == "gaussian":
            self.gaussian_sample(n_pts)
            radius = 30
        elif sampling_method == "bridge":
            self.bridge_sample(n_pts)
```

```
            radius = 70
        ### YOUR CODE HERE ###
        pairs = []
        node_list = []
        kdtree = spatial.KDTree(np.array(self.samples))
        # kd_tree pairs list
        kd_pairs = list(kdtree.query_pairs(radius))

        # All points in Kdtree are checked
        for query in kd_pairs:
            collision = self.check_collision(self.samples[query[0]],
self.samples[query[1]])
            if not collision:
                weight = self.dis(self.samples[query[0]],
self.samples[query[1]])
                if weight != 0:
                    pairs.append((query[0], query[1], weight))
                    node_list.append(query[0])
                    node_list.append(query[1])

        self.graph.add_nodes_from(node_list)
        self.graph.add_weighted_edges_from(pairs)

        # Print constructed graph information
        n_nodes = self.graph.number_of_nodes()
        n_edges = self.graph.number_of_edges()
        print("The constructed graph has %d nodes and %d edges" % (n_nodes,
n_edges))

    def search(self, start, goal):
        '''Search for a path in graph given start and goal location
        arguments:
            start - start point coordinate [row, col]
            goal - goal point coordinate [row, col]

        Temporary add start and goal node, edges of them and their nearest
neighbors
        to graph for self.graph to search for a path.
        '''
        # Clear previous path
        self.path = []

        # Temporarily add start and goal to the graph
        self.samples.append(start)
        self.samples.append(goal)
        # start and goal id will be 'start' and 'goal' instead of some
integer
        self.graph.add_nodes_from(['start', 'goal'])

        ### YOUR CODE HERE ###
        start_pairs = []
        goal_pairs = []
        # Setting up a goal radius
        radius = 100

        # Evaluation for start pairs:
        for i in range(len(self.samples)):
```

```
            # Calculating the starting point
            start = self.samples[len(self.samples) - 2]
            node = self.samples[i]
            if start != node:
                collision = self.check_collision(start, node)
                if not collision:
                    # calculate weight
                    weight = self.dis(start, node)
                    if weight != 0 and weight < radius:
                        start_pairs.append(('start',
self.samples.index(node), weight))

        # Evaluation for goal pairs:
        for j in range(len(self.samples)):
            # Calculating the goal point
            goal = self.samples[len(self.samples) - 1]
            node = self.samples[j]
            if goal != node:
                collision = self.check_collision(goal, node)
                if not collision:
                    # calculate weight
                    weight = self.dis(goal, node)
                    if weight != 0 and weight < radius:
                        goal_pairs.append(('goal', self.samples.index(node),
weight))

        self.graph.add_weighted_edges_from(start_pairs)
        self.graph.add_weighted_edges_from(goal_pairs)

        # Search using Dijkstra
        try:
            self.path =
nx.algorithms.shortest_paths.weighted.dijkstra_path(self.graph, 'start',
'goal')
            path_length =
nx.algorithms.shortest_paths.weighted.dijkstra_path_length(self.graph,
'start', 'goal')
            print("The path length is %.2f" % path_length)
        except nx.exception.NetworkXNoPath:
            print("No path found")

        # Draw result
        self.draw_map()

        # Remove start and goal node and their edges
        self.samples.pop(-1)
        self.samples.pop(-1)
        self.graph.remove_nodes_from(['start', 'goal'])
        self.graph.remove_edges_from(start_pairs)
        self.graph.remove_edges_from(goal_pairs)
```

RRT :

```
# Standard Algorithm Implementation
# Sampling-based Algorithms RRT and RRT*
```

```python
import matplotlib.pyplot as plt
import numpy as np
import math
# import bresenham

# Class for each tree node
class Node:
    def __init__(self, row, col):
        self.row = row         # coordinate
        self.col = col         # coordinate
        self.parent = None     # parent node
        self.cost = 0.0        # cost


# Class for RRT
class RRT:
    # Constructor
    def __init__(self, map_array, start, goal):
        self.map_array = map_array            # map array, 1->free, 0-
>obstacle
        self.size_row = map_array.shape[0]    # map size
        self.size_col = map_array.shape[1]    # map size

        self.start = Node(start[0], start[1]) # start node
        self.goal = Node(goal[0], goal[1])    # goal node
        self.vertices = []                    # list of nodes
        self.found = False                    # found flag


    def init_map(self):
        '''Intialize the map before each search
        '''
        self.found = False
        self.vertices = []
        self.vertices.append(self.start)


    def dis(self, node1, node2):
        '''Calculate the euclidean distance between two nodes
        arguments:
            node1 - node 1
            node2 - node 2

        return:
            euclidean distance between two nodes
        '''
        ### YOUR CODE HERE ###
        return np.sqrt((node1.row - node2.row)**2 + (node1.col -
node2.col)**2)


    def check_collision(self, node1, node2):
        '''Check if the path between two nodes collide with obstacles
        arguments:
            node1 - node 1
            node2 - node 2
```

```
        return:
            True if the new node is valid to be connected
        '''
        ### YOUR CODE HERE ###
        # x1, y1 = node1.row, node1.col
        # x2, y2 = node2.row, node2.col
        # points = int(list(bresenham.bresenham(x1, y1, x2, y2)))
        points = np.linspace([node1.row, node1.col], [node2.row, node2.col],
dtype=int)
        for p in points:
            x, y = p
            # colliding condition
            if self.map_array[x][y] == 0:
                return False
        return True


    def get_new_point(self, goal_bias):
        '''Choose the goal or generate a random point
        arguments:
            goal_bias - the possibility of choosing the goal instead of a
random point

        return:
            point - the new point
        '''
        ### YOUR CODE HERE ###
        rand_row = np.random.randint(0, self.size_row - 1)
        rand_col = np.random.randint(0, self.size_col - 1)
        rand_node = Node(rand_row, rand_col)
        # random point probability
        if np.random.rand() < goal_bias:
            return self.goal
        else:
            return rand_node


    def get_nearest_node(self, point):
        '''Find the nearest node in self.vertices with respect to the new
point
        arguments:
            point - the new point

        return:
            the nearest node
        '''
        ### YOUR CODE HERE ###
        minimum = 999999
        nearest_node = self.vertices[0]
        #  find minimum of all nodes
        for node in self.vertices:
            if self.dis(node, point) < minimum:
                nearest_node = node
                minimum = self.dis(node, point)
        return nearest_node

    def get_neighbors(self, new_node, neighbor_size):
```

```
        '''Get the neighbors that are within the neighbor distance from the
node
        arguments:
            new_node - a new node
            neighbor_size - the neighbor distance

        return:
            neighbors - a list of neighbors that are within the neighbor
distance
        '''
        ### YOUR CODE HERE ###
        neighbours = []
        for node in self.vertices:
            # distance condition
            if self.dis(node, new_node) < neighbor_size:
                # collision condition
                if self.check_collision(node, new_node):
                    neighbours.append(node)
        return neighbours
        # return [self.vertices[0]]


    def rewire(self, new_node, neighbors):
        '''Rewire the new node and all its neighbors
        arguments:
            new_node - the new node
            neighbors - a list of neighbors that are within the neighbor
distance from the node

        Rewire the new node if connecting to a new neighbor node will give
least cost.
        Rewire all the other neighbor nodes.
        '''
        ### YOUR CODE HERE ###
        distances = []
        for n in neighbors:
            distances.append(n.cost + self.dis(n, new_node))
        # get the best neighbour distance
        closest_neigh = neighbors[np.argmin(distances)]
        # make parent
        new_node.parent = closest_neigh
        # change cost
        new_node.cost = closest_neigh.cost + self.dis(new_node,
closest_neigh)
        neighbors.remove(closest_neigh)
        # rewire other neighbours
        for n in neighbors:
            dist_new_node = new_node.cost + self.dis(new_node, n)
            if n.cost > dist_new_node and not self.check_collision(new_node,
n):
                self.vertices.remove(n)
                n.parent = new_node
                n.cost = new_node.cost + dist_new_node
                self.vertices.append(n)
        self.vertices.append(new_node)
```

```
    def draw_map(self):
        '''Visualization of the result
        '''
        # Create empty map
        fig, ax = plt.subplots(1)
        img = 255 * np.dstack((self.map_array, self.map_array,
self.map_array))
        ax.imshow(img)

        # Draw Trees or Sample points
        for node in self.vertices[1:-1]:
            plt.plot(node.col, node.row, markersize=3, marker='o', color='y')
            plt.plot([node.col, node.parent.col], [node.row,
node.parent.row], color='y')

        # Draw Final Path if found
        if self.found:
            cur = self.goal
            while cur.col != self.start.col or cur.row != self.start.row:
                plt.plot([cur.col, cur.parent.col], [cur.row,
cur.parent.row], color='b')
                cur = cur.parent
                plt.plot(cur.col, cur.row, markersize=3, marker='o',
color='b')

        # Draw start and goal
        plt.plot(self.start.col, self.start.row, markersize=5, marker='o',
color='g')
        plt.plot(self.goal.col, self.goal.row, markersize=5, marker='o',
color='r')

        # show image
        plt.show()


    def RRT(self, n_pts=1000):
        '''RRT main search function
        arguments:
            n_pts - number of points try to sample,
                    not the number of final sampled points

        In each step, extend a new node if possible, and check if reached the
goal
        '''
        # Remove previous result
        self.init_map()
        ### YOUR CODE HERE ###
        # In each step,
        goal_bias = 0.05
        for i in range(n_pts):
            # get a new point,
            curr_node = self.get_new_point(goal_bias)
            # get its nearest node,
            near_node = self.get_nearest_node(curr_node)
            # extension
            extend_dis = 10
```

```
            theta = math.atan2((curr_node.col - near_node.col),
(curr_node.row - near_node.row))
            # extend the node and check collision to decide whether to add or
drop,
            new_row = int(near_node.row + extend_dis * (math.cos(theta)))
            new_col = int(near_node.col + extend_dis * (math.sin(theta)))
            new_node = Node(new_row, new_col)
            if (0 <= new_row < self.size_row) and (0 <= new_col <
self.size_col) \
                    and self.check_collision(new_node, near_node):
                new_node.parent = near_node
                new_node.cost = extend_dis + near_node.cost
                self.vertices.append(new_node)
                if not self.found:
                    short_dis = self.dis(self.goal, new_node)
                    # and check if reach the neighbor region of the goal if
the path is not found.
                    if short_dis < extend_dis and
self.check_collision(self.goal, new_node):  # in neighbourhood
                        self.goal.parent = new_node
                        self.found = True
                        self.goal.cost = short_dis + new_node.cost
                        self.vertices.append(self.goal)
                        break
        # Output
        if self.found:
            steps = len(self.vertices) - 2
            length = self.goal.cost
            print("It took %d nodes to find the current path" %steps)
            print("The path length is %.2f" %length)
        else:
            print("No path found")

        # Draw result
        self.draw_map()


    def RRT_star(self, n_pts=1000, neighbor_size=40):
        '''RRT* search function
        arguments:
            n_pts - number of points try to sample,
                    not the number of final sampled points
            neighbor_size - the neighbor distance

        In each step, extend a new node if possible, and rewire the node and
its neighbors
        '''
        # Remove previous result
        self.init_map()

        ### YOUR CODE HERE ###
        # In each step,
        goal_bias = 0.05
        for i in range(n_pts):
            # get a new point,
            curr_node = self.get_new_point(goal_bias)
            # get its nearest node,
```

```
            near_node = self.get_nearest_node(curr_node)
            # extension
            extend_dis = 10
            theta = math.atan2((curr_node.col - near_node.col),
(curr_node.row - near_node.row))
            # extend the node and check collision to decide whether to add or
drop,
            new_row = int(near_node.row + extend_dis * (math.cos(theta)))
            new_col = int(near_node.col + extend_dis * (math.sin(theta)))
            new_node = Node(new_row, new_col)
            if (0 <= new_row < self.size_row) and (0 <= new_col <
self.size_col) \
                    and self.check_collision(new_node, near_node):
                # find neighbours
                neighbours = self.get_neighbors(new_node, neighbor_size)
                # rewire them
                self.rewire(new_node, neighbours)
                if not self.found:
                    short_dis = self.dis(self.goal, new_node)
                    # and check if reach the neighbor region of the goal if
the path is not found.
                    if short_dis < extend_dis and
self.check_collision(self.goal, new_node):  # in neighbourhood
                        self.goal.parent = new_node
                        self.found = True
                        self.goal.cost = short_dis + new_node.cost
                        self.vertices.append(self.goal)

        # Output
        if self.found:
            steps = len(self.vertices) - 2
            length = self.goal.cost
            print("It took %d nodes to find the current path" %steps)
            print("The path length is %.2f" %length)
        else:
            print("No path found")

        # Draw result
        self.draw_map()
```