```
import math
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial import KDTree #gerekli libraryler indirilir

# Parameters
N_SAMPLE = 500  # Number of sample points
N_KNN = 10  # Number of edges from one sampled point
MAX_EDGE_LEN = 30.0  # [m] Maksimum edge uzunuğu

show_animation = True

class Node:
    """
    Node class for Dijkstra search
    """
    def __init__(self, x, y, cost, parent_index):
        self.x = x
        self.y = y
        self.cost = cost
        self.parent_index = parent_index

    def __str__(self):
        return str(self.x) + "," + str(self.y) + "," +\
               str(self.cost) + "," + str(self.parent_index)

def prm_planning(start_x, start_y, goal_x, goal_y,
                 obstacle_x_list, obstacle_y_list, robot_radius, *, rng=None):
    """
    Run probabilistic road map planning

    :param start_x: start x position
    :param start_y: start y position
    :param goal_x: goal x position
    :param goal_y: goal y position
    :param obstacle_x_list: obstacle x positions
    :param obstacle_y_list: obstacle y positions
    :param robot_radius: robot radius
    :param rng: (Optional) Random generator
    :return: Path coordinates
    """
    # Create a KDTree for obstacles
    obstacle_kd_tree = KDTree(np.vstack((obstacle_x_list, obstacle_y_list)).T)
```

```
    # Sample points in the environment
    sample_x, sample_y = sample_points(start_x, start_y, goal_x, goal_y,
                                       robot_radius,
                                       obstacle_x_list, obstacle_y_list,
                                       obstacle_kd_tree, rng)
    if show_animation:
        plt.plot(sample_x, sample_y, ".b")

    # Generate road map from the sampled points
    road_map = generate_road_map(sample_x, sample_y,
                                 robot_radius, obstacle_kd_tree)

    # Find the path using Dijkstra's algorithm
    rx, ry = dijkstra_planning(
        start_x, start_y, goal_x, goal_y, road_map, sample_x, sample_y)

    return rx, ry

def is_collision(sx, sy, gx, gy, rr, obstacle_kd_tree):
    """
    Check if the path between (sx, sy) and (gx, gy) collides with any obstacle
    """
    x = sx
    y = sy
    dx = gx - sx
    dy = gy - sy
    yaw = math.atan2(gy - sy, gx - sx)
    d = math.hypot(dx, dy)

    if d >= MAX_EDGE_LEN:
        return True

    D = rr
    n_step = round(d / D)

    for i in range(n_step):
        dist, _ = obstacle_kd_tree.query([x, y])
        if dist <= rr:
            return True  # Collision
        x += D * math.cos(yaw)
        y += D * math.sin(yaw)

    # Check goal point
    dist, _ = obstacle_kd_tree.query([gx, gy])
```

```
    if dist <= rr:
        return True  # Collision

    return False  # No collision

def generate_road_map(sample_x, sample_y, rr, obstacle_kd_tree):
    """
    Generate road map from sampled points

    sample_x: [m] x positions of sampled points
    sample_y: [m] y positions of sampled points
    robot_radius: Robot radius [m]
    obstacle_kd_tree: KDTree object of obstacles
    """
    road_map = []
    n_sample = len(sample_x)
    sample_kd_tree = KDTree(np.vstack((sample_x, sample_y)).T)

    for (i, ix, iy) in zip(range(n_sample), sample_x, sample_y):
        dists, indexes = sample_kd_tree.query([ix, iy], k=n_sample)
        edge_id = []

        for ii in range(1, len(indexes)):
            nx = sample_x[indexes[ii]]
            ny = sample_y[indexes[ii]]

            if not is_collision(ix, iy, nx, ny, rr, obstacle_kd_tree):
                edge_id.append(indexes[ii])

            if len(edge_id) >= N_KNN:
                break

        road_map.append(edge_id)

    return road_map

def dijkstra_planning(sx, sy, gx, gy, road_map, sample_x, sample_y):
    """
    Run Dijkstra's algorithm to find the shortest path

    sx: start x position [m]
    sy: start y position [m]
    gx: goal x position [m]
    gy: goal y position [m]
```

```
road_map: Generated road map
sample_x: Sampled x positions
sample_y: Sampled y positions
@return: Path coordinates ([x1, x2, ...], [y1, y2, ...])
"""
start_node = Node(sx, sy, 0.0, -1)
goal_node = Node(gx, gy, 0.0, -1)

open_set, closed_set = dict(), dict()
open_set[len(road_map) - 2] = start_node

path_found = True

while True:
    if not open_set:
        print("Cannot find path")
        path_found = False
        break

    c_id = min(open_set, key=lambda o: open_set[o].cost)
    current = open_set[c_id]

    if show_animation and len(closed_set.keys()) % 2 == 0:
        plt.gcf().canvas.mpl_connect(
            'key_release_event',
            lambda event: [exit(0) if event.key == 'escape' else None])
        plt.plot(current.x, current.y, "xg")
        plt.pause(0.001)

    if c_id == (len(road_map) - 1):
        print("Goal is found!")
        goal_node.parent_index = current.parent_index
        goal_node.cost = current.cost
        break

    del open_set[c_id]
    closed_set[c_id] = current

    for i in range(len(road_map[c_id])):
        n_id = road_map[c_id][i]
        dx = sample_x[n_id] - current.x
        dy = sample_y[n_id] - current.y
        d = math.hypot(dx, dy)
        node = Node(sample_x[n_id], sample_y[n_id],
```

```
                    current.cost + d, c_id)

            if n_id in closed_set:
                continue
            if n_id in open_set:
                if open_set[n_id].cost > node.cost:
                    open_set[n_id].cost = node.cost
                    open_set[n_id].parent_index = c_id
            else:
                open_set[n_id] = node

    if not path_found:
        return [], []

    rx, ry = [goal_node.x], [goal_node.y]
    parent_index = goal_node.parent_index
    while parent_index != -1:
        n = closed_set[parent_index]
        rx.append(n.x)
        ry.append(n.y)
        parent_index = n.parent_index

    return rx, ry

def plot_road_map(road_map, sample_x, sample_y):
    for i, _ in enumerate(road_map):
        for ii in range(len(road_map[i])):
            ind = road_map[i][ii]
            plt.plot([sample_x[i], sample_x[ind]],
                     [sample_y[i], sample_y[ind]], "-k")

def sample_points(sx, sy, gx, gy, rr, ox, oy, obstacle_kd_tree, rng):
    max_x = max(ox)
    max_y = max(oy)
    min_x = min(ox)
    min_y = min(oy)

    sample_x, sample_y = [], []

    if rng is None:
        rng = np.random.default_rng()

    while len(sample_x) <= N_SAMPLE:
        tx = (rng.random() * (max_x - min_x)) + min_x
```

```
        ty = (rng.random() * (max_y - min_y)) + min_y

        dist, index = obstacle_kd_tree.query([tx, ty])
        if dist >= rr:
            sample_x.append(tx)
            sample_y.append(ty)

    sample_x.append(sx)
    sample_y.append(sy)
    sample_x.append(gx)
    sample_y.append(gy)

    return sample_x, sample_y

def main(rng=None):
    print(__file__ + " start!!")

    # Start and goal position
    sx = 10.0  # [m]
    sy = 10.0  # [m]
    gx = 50.0  # [m]
    gy = 50.0  # [m]
    robot_size = 5.0  # [m]

    # Define obstacle coordinates
    ox, oy = [], []
    for i in range(60):
        ox.append(i)
        oy.append(0.0)
    for i in range(60):
        ox.append(60.0)
        oy.append(i)
    for i in range(61):
        ox.append(i)
        oy.append(60.0)
    for i in range(61):
        ox.append(0.0)
        oy.append(i)
    for i in range(40):
        ox.append(20.0)
        oy.append(i)
```

“”

Probabilistik Yol Haritası (PRM) planlayıcısı, robotik ve otonom araçlar için kullanılan bir yol planlama algoritmasıdır. Bu algoritma, robotun başlangıç ve hedef noktaları arasında engellerden kaçınarak bir yol bulmasını sağlar. Kodda, rastgele örnekleme yöntemiyle belirli sayıda nokta seçilir ve bu noktalar arasında yollar oluşturulur. Engellerin etrafında güvenli yollar belirlemek için KDTree kullanılır ve Dijkstra algoritmasıyla en kısa yol bulunur.

Kodun detayları şu şekildedir:

- **Örnekleme**: Rastgele noktalar seçilir ve bu noktalar arasında yollar oluşturulur. Örnekleme sırasında, robotun güvenli bir mesafede olması için engellere yakın noktalar elenir.
- **Yol Haritası Oluşturma**: Seçilen noktalar arasında kenarlar (yollar) oluşturulur. Bu kenarlar, engellerle çakışmadığı sürece kabul edilir.
- **Dijkstra Algoritması**: Başlangıç ve hedef noktaları arasındaki en kısa yolu bulmak için kullanılır. Bu algoritma, en düşük maliyetli yolu seçerek ilerler.

Bu algoritmanın üstünlüğü, engelli ve karmaşık ortamlarda bile etkili bir yol bulma yeteneğine sahip olmasıdır. Rastgele örnekleme ve Dijkstra algoritmasının kombinasyonu, büyük ve karmaşık haritalarda bile hızlı ve güvenilir sonuçlar sağlar. Ayrıca, bu yöntem esnektir ve farklı engel düzenlerine uyarlanabilir, bu da onu gerçek dünya uygulamaları için ideal kılar. Engellerden kaçınma, çeşitli yollar oluşturma ve en kısa yolu bulma yetenekleri, PRM algoritmasını çok yönlü ve güçlü bir araç haline getirir.
“”