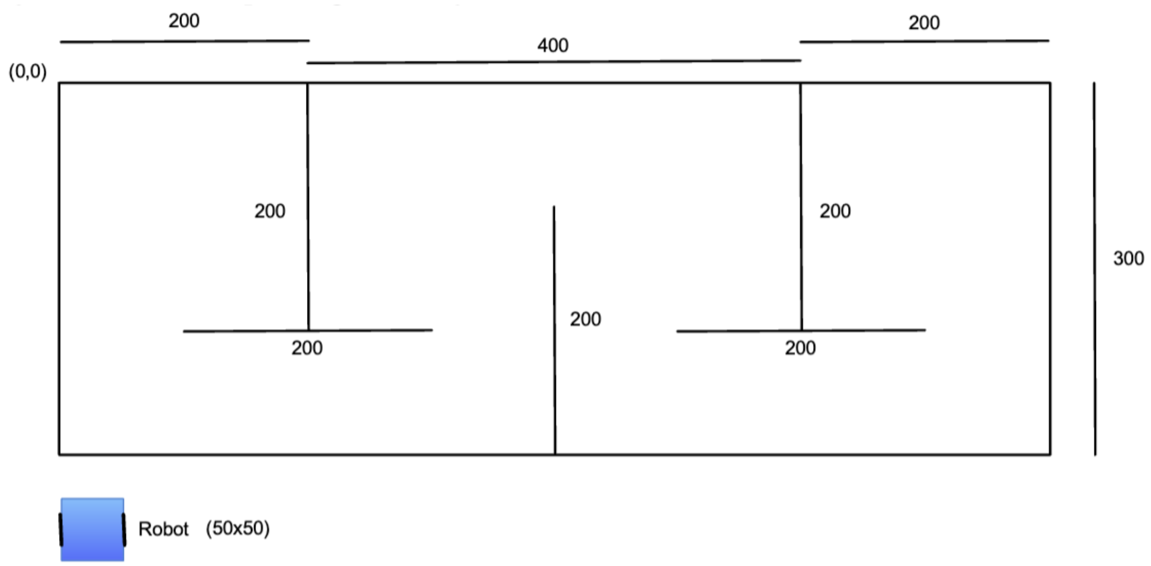
The world model is shown in figure 1. The robot is a differential drive system with a square geometry of size 50×50.



1 Question 1

Generate the configuration space for the robot with a grid size of 2×2 and 5 deg in angular resolution. Generate an illustration of what the configuration space looks like with the robot at orientations 0, 45 and 90 deg.

1.1 Method

To make visualization easier, I reduced the original grid size from 2×2 to 1×1. As a result, everything in the world shrinks accordingly. The map is now 400×150, and the robot size is adjusted to 25×25 to ensure smooth performance.To generate the configuration space (C-space) for the robot, I first calculate the Minkowski sum of the obstacle, wall, and robot, then compute the convex hull to form a new polygon. However, a C-space with a 5° resolution would require $180/5 = 36$ different orientations. To simplify this, I assume the robot is circular with a radius of $12.5\sqrt{2}$ and round it up to 18 for a conservative approximation. For the configuration space at orientations $0°, 45°, 90°$,I rotate the robot to the specified angle and apply the Minkowski sum method as described above.

1.2 Code

```
    import numpy as np
import matplotlib.pyplot as plt
from shapely.geometry import LineString, Polygon, Point
from shapely.affinity import rotate, translate

# Define the simplified world and robot
WORLD_WIDTH = 400  #
WORLD_HEIGHT = 150
OBSTACLES = [# Define a list of obstacles
    LineString([(50,100),(150,100)]),
    LineString([(100,0), (100, 100)]),
    LineString([(200,50), (200, 150)]),
    LineString([(250,100),(350,100)]),
    LineString([(300,0),(300,100)]),
    LineString([(0,0),(0,150)]),  # Left wall
    LineString([(0,0),(400,0)]),  # Top wall
```

```
    LineString([(400,0),(400,150)]), # Right wall
    LineString([(0,150),(400,150)])  # Bottom wall
]

ROBOT_RADIUS = 18 # Define the radius of the circular robot
# Generate a circular robot using a buffer around a point
ROBOT_circle = Point(0, 0).buffer(ROBOT_RADIUS)

ROBOT_SIZE = 25# Define the size of the robot
ROBOT = Polygon([(-ROBOT_SIZE/2, -ROBOT_SIZE/2),  # Define the shape of the robot (a square)
                  (ROBOT_SIZE/2, -ROBOT_SIZE/2),
                  (ROBOT_SIZE/2, ROBOT_SIZE/2),
                  (-ROBOT_SIZE/2, ROBOT_SIZE/2)])

# Calculate Minkowski sum
def minkowski_sum(obstacle, robot):  # Define the minkowski_sum function to calculate the Minkowski
    robot_points = np.array(robot.exterior.coords)  # Get the coordinates of the robot polygon
    obstacle_points = np.array(obstacle.coords)  # Get the coordinates of the obstacle polygon
    minkowski_points = []  # Initialize a list to hold the Minkowski sum points


    for rp in robot_points:  # Iterate over each point of the robot
        for op in obstacle_points:  # Iterate over each point of the obstacle
            minkowski_points.append(rp + op)  # Add the robot's point and obstacle's point to get Mi


    minkowski_poly = Polygon(minkowski_points).convex_hull  # Construct a polygon from the Minkowski
    return minkowski_poly  # Return the Minkowski sum polygon

# Generate configuration space
def generate_cspace(obstacles, robot, angle_resolution=5):  # Define the generate_cspace function to
    cspaces = []  # Initialize a list to hold the configuration spaces
    for angle in range(0, 180, angle_resolution):  # Iterate over angles from 0 to 180 with the spec
        rotated_robot = rotate(robot, angle, origin=(0,0), use_radians=False)  # Rotate the robot by
        cspace_polys = [minkowski_sum(ob, rotated_robot) for ob in obstacles]  # Calculate Minkowski
        cspaces.append((angle, cspace_polys))  # Append the angle and corresponding configuration sp
    return cspaces  # Return the list of configuration spaces

# Plotting function to visualize configuration spaces
def plot_cspace(angle,cspaces, world_width, world_height):  # Define the plot_cspace function to plo
        plt.figure(figsize=(13, 5))
        plt.title(f"C-Space at {angle}°")  # Set the title of each subplot
        plt.xlim(0, world_width)  # Set the x-axis range
        plt.ylim(world_height,0 )  # Set the y-axis range

        # Set the x and y axis ticks
        x_ticks = np.arange(0, world_width + 10, 10)
        y_ticks = np.arange(0, world_height + 10, 10)
        plt.xticks(x_ticks)
        plt.yticks(y_ticks)

        # Format the tick labels
        plt.gca().xaxis.set_ticks_position('top')  # Set the tick positions of the x-axis to the top
        plt.gca().xaxis.set_major_formatter(plt.FormatStrFormatter('%d'))
        plt.gca().yaxis.set_major_formatter(plt.FormatStrFormatter('%d'))

        plt.tick_params(axis='both', which='major', labelsize=8)  # Set the font size of the tick la
        plt.grid(which='major', color='gray', linestyle='-', linewidth=0.5)  # Draw grid lines
```

```
        for ob in OBSTACLES:  # Iterate over obstacles
            x, y = ob.xy  # Get the x and y coordinates of the obstacle
            plt.plot(x, y, color='gray', linewidth=2)  # Plot the obstacle in gray

        for cspace_poly in cspaces[angle // 5][1]:  # Iterate over the configuration space polygons
            x, y = cspace_poly.exterior.xy  # Get the x and y coordinates of the configuration space
            plt.fill(x, y, color='blue', alpha=0.3)  # Plot the configuration space in blue with tra

        plt.tight_layout()  # Adjust the layout of subplots to avoid overlapping
        plt.show()  # Display the plot

# Main program
cspace1=generate_cspace(OBSTACLES, ROBOT_circle)
#for angle in range(0, 180,5):
#    plot_cspace(angle,cspace1, WORLD_WIDTH, WORLD_HEIGHT)
cspaces2 = generate_cspace(OBSTACLES, ROBOT)  # Generate the configuration space
for angle in [0, 45, 90]:
    plot_cspace(angle,cspaces2, WORLD_WIDTH, WORLD_HEIGHT)  # Plot the configuration space
```

1.3 Results

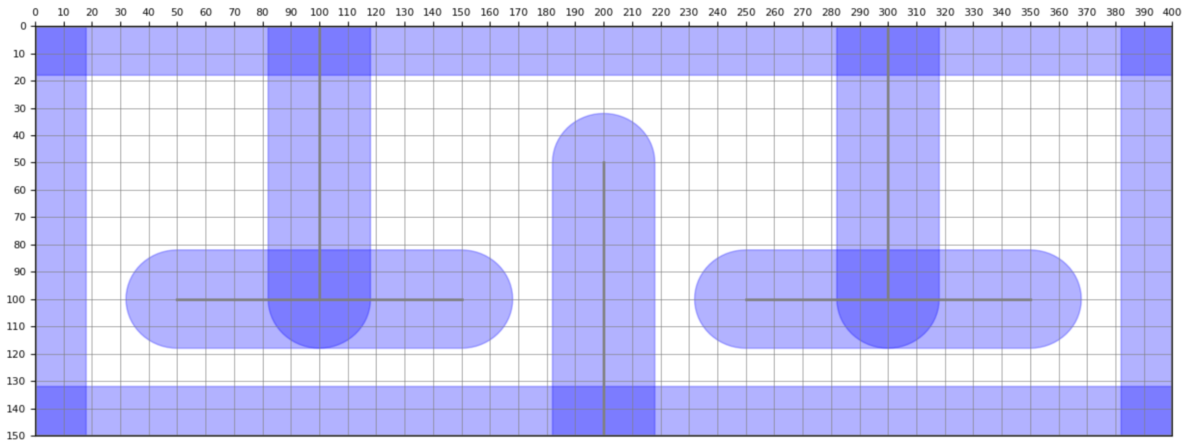


Figure 1: C_Space

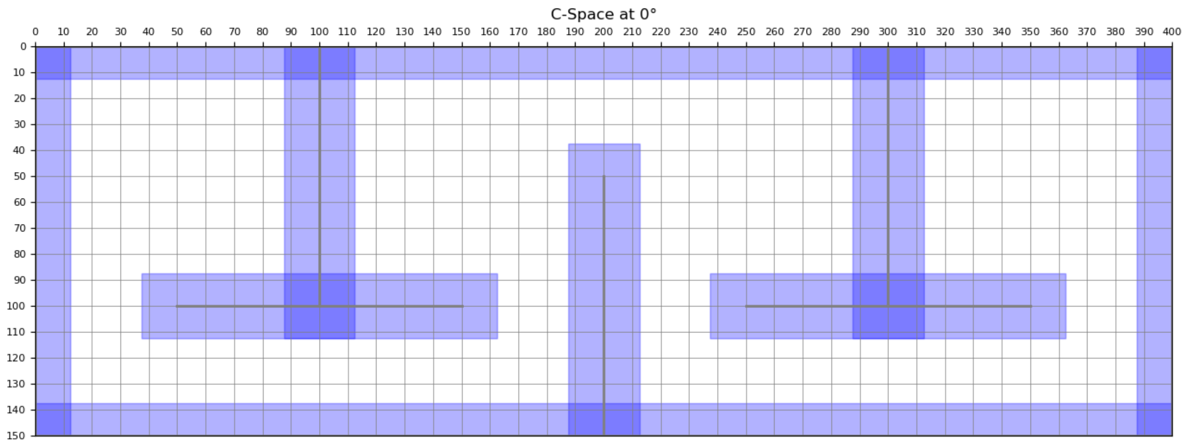


Figure 2: C_Space_0 degree

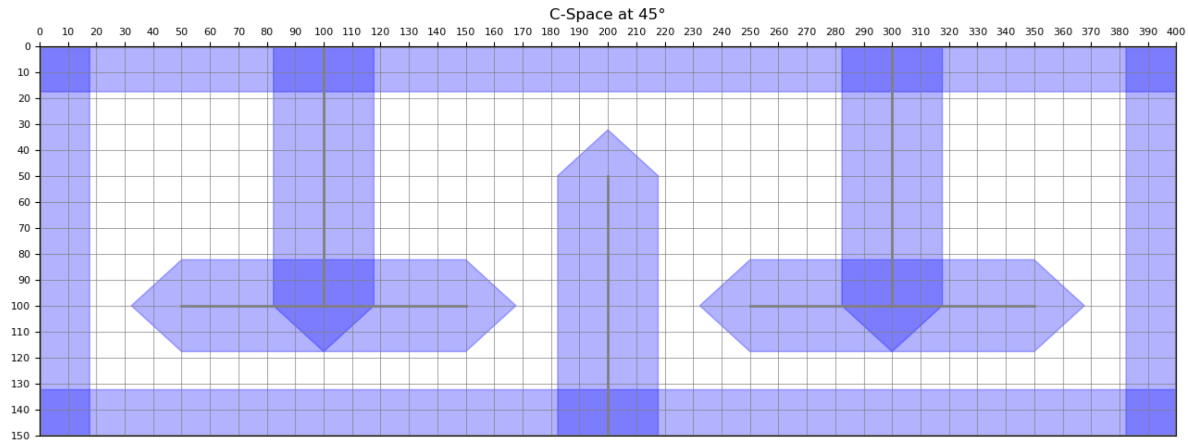


Figure 3: C_Space_45 degree

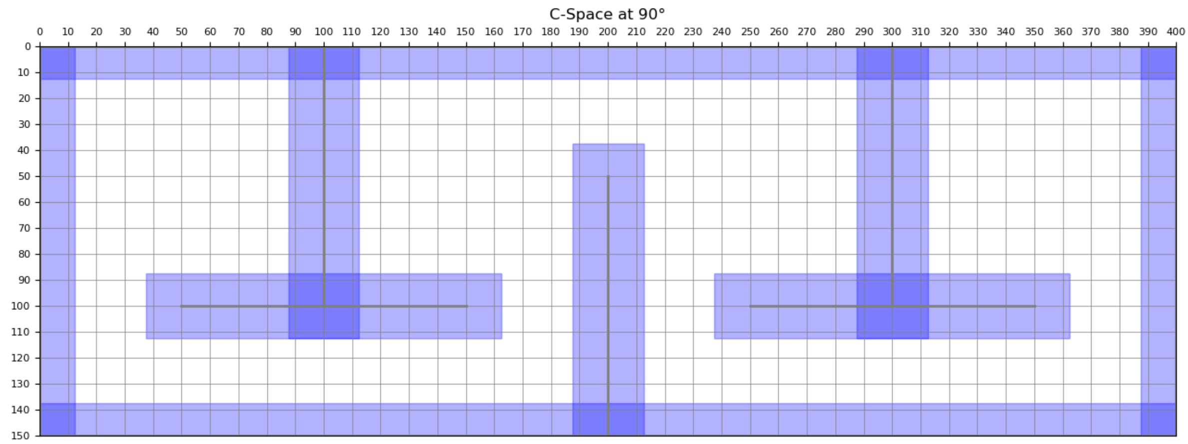


Figure 4: C_Space_90 degree

2 Question 2

Use greedy search to find the shortest path between start-point (50,50) and end-point (750,50). Illustrate the path and provide its length.

2.1 Method

To find the shortest path between start-point and end-point, I use Dijkstra algorithm. The algorithm works by maintaining a set of tentative distances to each node, continuously updating these distances as it explores outward from the source node, and always selecting the unvisited node with the smallest tentative distance as the next node to explore. To compute the path length, I use the Euclidean distance formula. Based on this computation, the length of the path is approximately 632.31 grids.

2.2 Code

```
    import matplotlib.pyplot as plt
from shapely.geometry import box
import math
import numpy as np

show_animation = True
```

```
class Dijkstra:

    def __init__(self, ox, oy, resolution, robot_size):
        self.min_x = None
        self.min_y = None
        self.max_x = None
        self.max_y = None
        self.x_width = None
        self.y_width = None
        self.obstacle_map = None

        self.resolution = resolution
        self.robot_size = robot_size
        self.calc_obstacle_map(ox, oy)
        self.motion = self.get_motion_model()

    class Node:
        def __init__(self, x, y, cost, parent_index):
            self.x = x  # index of grid
            self.y = y  # index of grid
            self.cost = cost
            self.parent_index = parent_index  # index of previous Node

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

    def calc_obstacle_map(self, ox, oy):

        self.min_x = round(min(ox))
        self.min_y = round(min(oy))
        self.max_x = round(max(ox))
        self.max_y = round(max(oy))
        print("min_x:", self.min_x)
        print("min_y:", self.min_y)
        print("max_x:", self.max_x)
        print("max_y:", self.max_y)

        self.x_width = round((self.max_x - self.min_x) / self.resolution)
        self.y_width = round((self.max_y - self.min_y) / self.resolution)
        print("x_width:", self.x_width)
        print("y_width:", self.y_width)

        # obstacle map generation
        self.obstacle_map = [[False for _ in range(self.y_width)]
                             for _ in range(self.x_width)]

        half_size = self.robot_size / 2.0

        for ix in range(self.x_width):
            x = self.calc_position(ix, self.min_x)
            for iy in range(self.y_width):
                y = self.calc_position(iy, self.min_y)

                #robot_box = box(x - half_size, y - half_size,
                #               x + half_size, y + half_size)
                for ox_i, oy_i in zip(ox, oy):
```

```
                if (abs(ox_i - x) <= half_size and
                    abs(oy_i - y) <= half_size):
                    self.obstacle_map[ix][iy] = True
                    break

def planning(self, sx, sy, gx, gy):
    """
    dijkstra path search

    input:
        s_x: start x position [m]
        s_y: start y position [m]
        gx: goal x position [m]
        gx: goal x position [m]

    output:
        rx: x position list of the final path
        ry: y position list of the final path
    """

    start_node = self.Node(self.calc_xy_index(sx, self.min_x),
                           self.calc_xy_index(sy, self.min_y), 0.0, -1)
    goal_node = self.Node(self.calc_xy_index(gx, self.min_x),
                          self.calc_xy_index(gy, self.min_y), 0.0, -1)

    open_set, closed_set = dict(), dict()
    open_set[self.calc_index(start_node)] = start_node

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

        # show graph
        if show_animation:  # pragma: no cover
            plt.plot(self.calc_position(current.x, self.min_x),
                     self.calc_position(current.y, self.min_y), "xc")
            # for stopping simulation with the esc key.
            plt.gcf().canvas.mpl_connect(
                'key_release_event',
                lambda event: [exit(0) if event.key == 'escape' else None])
            if len(closed_set.keys()) % 10 == 0:
                plt.pause(0.001)

        if current.x == goal_node.x and current.y == goal_node.y:
            print("Find goal")
            goal_node.parent_index = current.parent_index
            goal_node.cost = current.cost
            break

        # Remove the item from the open set
        del open_set[c_id]

        # Add it to the closed set
        closed_set[c_id] = current

        # expand search grid based on motion model
        for move_x, move_y, move_cost in self.motion:
            node = self.Node(current.x + move_x,
```

```
                              current.y + move_y,
                              current.cost + move_cost, c_id)
            n_id = self.calc_index(node)

            if n_id in closed_set:
                continue

            if not self.verify_node(node):
                continue

            if n_id not in open_set:
                open_set[n_id] = node  # Discover a new node
            else:
                if open_set[n_id].cost >= node.cost:
                    # This path is the best until now. record it!
                    open_set[n_id] = node

    rx, ry = self.calc_final_path(goal_node, closed_set)

    return rx, ry

def calc_final_path(self, goal_node, closed_set):
    # generate final course
    rx, ry = [self.calc_position(goal_node.x, self.min_x)], [
        self.calc_position(goal_node.y, self.min_y)]
    parent_index = goal_node.parent_index
    while parent_index != -1:
        n = closed_set[parent_index]
        rx.append(self.calc_position(n.x, self.min_x))
        ry.append(self.calc_position(n.y, self.min_y))
        parent_index = n.parent_index

    return rx, ry

def calc_position(self, index, minp):
    pos = index * self.resolution + minp
    return pos

def calc_xy_index(self, position, minp):
    return round((position - minp) / self.resolution)

def calc_index(self, node):
    return (node.y - self.min_y) * self.x_width + (node.x - self.min_x)

def verify_node(self, node):
    px = self.calc_position(node.x, self.min_x)
    py = self.calc_position(node.y, self.min_y)

    if px < self.min_x:
        return False
    if py < self.min_y:
        return False
    if px >= self.max_x:
        return False
    if py >= self.max_y:
        return False

    if self.obstacle_map[node.x][node.y]:
```

```
            return False

        return True




    @staticmethod
    def get_motion_model():
        # dx, dy, cost
        motion = [[1, 0, 1],
                  [0, 1, 1],
                  [-1, 0, 1],
                  [0, -1, 1],
                  [-1, -1, math.sqrt(2)],
                  [-1, 1, math.sqrt(2)],
                  [1, -1, math.sqrt(2)],
                  [1, 1, math.sqrt(2)]]

        return motion

    def cal_euclidean_dist(self,rx,ry):
        dist = 0
        for i in range(len(rx)-1):
            point1 = np.array([rx[i], ry[i]])
            point2 = np.array([rx[i+1], ry[i+1]])
            dist = dist + np.linalg.norm(point1 - point2)
        return dist



def main():
    print(__file__ + " start!!")

     # start and goal position
    sx = 25.0
    sy = 25.0
    gx = 375.0
    gy = 25.0
    grid_size = 1.0
    robot_size = 25

    # set obstacle positions
    ox, oy = [], []
    for i in range(50, 151):
        ox.append(i)
        oy.append(100.0)
    for i in range(0, 101):
        ox.append(100.0)
        oy.append(i)
    for i in range(50, 151):
        ox.append(200.0)
        oy.append(i)
    for i in range(250, 351):
        ox.append(i)
        oy.append(100.0)
    for i in range(0, 101):
        ox.append(300.0)
```

```
        oy.append(i)
    for i in range(0, 151):
        ox.append(0.0)
        oy.append(i)
    for i in range(0, 401):
        ox.append(i)
        oy.append(0.0)
    for i in range(0, 151):
        ox.append(400.0)
        oy.append(i)
    for i in range(0, 401):
        ox.append(i)
        oy.append(150.0)

    if show_animation:  # pragma: no cover
        plt.plot(ox, oy, ".k")
        plt.plot(sx, sy, "og")
        plt.plot(gx, gy, "xb")
        plt.grid(True)
        plt.axis("equal")
        plt.gca().xaxis.set_ticks_position('top')
        plt.xlim(0, 400)  # Set the x-axis range
        plt.ylim(150,0 )  # Set the y-axis range

    dijkstra = Dijkstra(ox, oy, grid_size, robot_size)
    rx, ry = dijkstra.planning(sx, sy, gx, gy)
    dist = dijkstra.cal_euclidean_dist(rx, ry)
    print("distance: ", dist)
    if show_animation:  # pragma: no cover
        plt.plot(rx, ry, "-r")
        plt.pause(0.01)
        plt.show()


if __name__ == '__main__':
    main()
```

2.3 Results

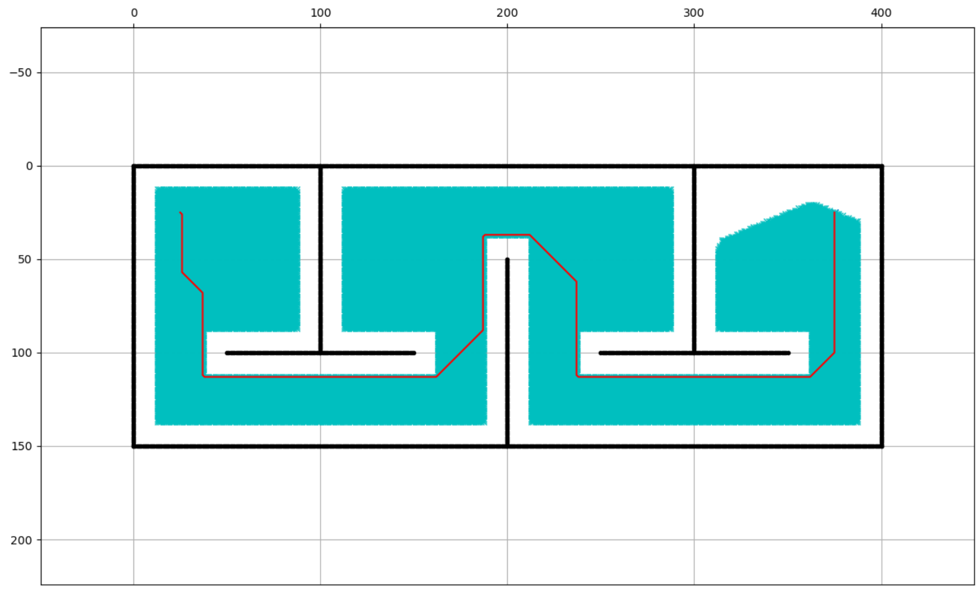


Figure 5: shortest path

3 Question 3

Compute the safest path from start to finish (hint: medial axis transform/Voronoi). Illustrate the path and provide its length.

3.1 Method

To compute the safest path from start to finish, I apply the Voronoi diagram to generate a safe roadmap and use the Dijkstra algorithm to find the optimal path. A Voronoi diagram is a fundamental geometric structure that partitions a space into regions based on proximity to specified points.As shown in Figure 6, the blue straight lines represent the edges of the Voronoi diagram. These edges are equidistant from the nearest obstacles (walls and barriers). The green curved line represents the actual "safest path" that the robot should follow. It is considered the "safest" because it maximizes the distance from all obstacles by following the medial axis between them. Additionally, I use the Euclidean distance formula to compute the length of the path which is approximately 777.9 grids.

3.2 Code

3.2.1 Voronoi road map

```
import math
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial import cKDTree, Voronoi
import sys
import pathlib
sys.path.append(str(pathlib.Path(__file__).parent.parent))

from Dijkstra_Search import DijkstraSearch
```

```
show_animation = True


class VoronoiRoadMapPlanner:

    def __init__(self):
        # parameter
        self.N_KNN = 10  # number of edge from one sampled point
        self.MAX_EDGE_LEN = 30.0  # [m] Maximum edge length

    def planning(self, sx, sy, gx, gy, ox, oy, robot_radius):
        obstacle_tree = cKDTree(np.vstack((ox, oy)).T)

        sample_x, sample_y = self.voronoi_sampling(sx, sy, gx, gy, ox, oy)
        if show_animation:  # pragma: no cover
            plt.plot(sample_x, sample_y, ".b")

        road_map_info = self.generate_road_map_info(
            sample_x, sample_y, robot_radius, obstacle_tree)

        rx, ry = DijkstraSearch(show_animation).search(sx, sy, gx, gy,
                                                       sample_x, sample_y,
                                                       road_map_info)
        return rx, ry

    def is_collision(self, sx, sy, gx, gy, rr, obstacle_kd_tree):
        x = sx
        y = sy
        dx = gx - sx
        dy = gy - sy
        yaw = math.atan2(gy - sy, gx - sx)
        d = math.hypot(dx, dy)

        if d >= self.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)

        # goal point check
        dist, _ = obstacle_kd_tree.query([gx, gy])
        if dist <= rr:
            return True  # collision

        return False  # OK

    def generate_road_map_info(self, node_x, node_y, rr, obstacle_tree):
        """
        Road map generation
```

```
    node_x: [m] x positions of sampled points
    node_y: [m] y positions of sampled points
    rr: Robot Radius[m]
    obstacle_tree: KDTree object of obstacles
    """

    road_map = []
    n_sample = len(node_x)
    node_tree = cKDTree(np.vstack((node_x, node_y)).T)

    for (i, ix, iy) in zip(range(n_sample), node_x, node_y):

        dists, indexes = node_tree.query([ix, iy], k=n_sample)

        edge_id = []

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

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

            if len(edge_id) >= self.N_KNN:
                break

        road_map.append(edge_id)

    #  plot_road_map(road_map, sample_x, sample_y)

    return road_map
def cal_euclidean_dist(self,rx,ry):
    dist = 0
    for i in range(len(rx)-1):
        point1 = np.array([rx[i], ry[i]])
        point2 = np.array([rx[i+1], ry[i+1]])
        dist = dist + np.linalg.norm(point1 - point2)
    return dist



@staticmethod
def plot_road_map(road_map, sample_x, sample_y):  # pragma: no cover

    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")

@staticmethod
def voronoi_sampling(sx, sy, gx, gy, ox, oy):
    oxy = np.vstack((ox, oy)).T

    # generate voronoi point
    vor = Voronoi(oxy)
    sample_x = [ix for [ix, _] in vor.vertices]
```

```
        sample_y = [iy for [_, iy] in vor.vertices]

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

        return sample_x, sample_y


def main():
    print(__file__ + " start!!")

    # start and goal position
    sx = 25.0  # [m]
    sy = 25.0  # [m]
    gx = 375.0  # [m]
    gy = 25.0  # [m]
    robot_size = 18  # [m]

    ox, oy = [], []
    for i in range(50, 151):
        ox.append(i)
        oy.append(100.0)
    for i in range(0, 101):
        ox.append(100.0)
        oy.append(i)
    for i in range(50, 151):
        ox.append(200.0)
        oy.append(i)
    for i in range(250, 351):
        ox.append(i)
        oy.append(100.0)
    for i in range(0, 101):
        ox.append(300.0)
        oy.append(i)
    for i in range(0, 151):
        ox.append(0.0)
        oy.append(i)
    for i in range(0, 401):
        ox.append(i)
        oy.append(0.0)
    for i in range(0, 151):
        ox.append(400.0)
        oy.append(i)
    for i in range(0, 401):
        ox.append(i)
        oy.append(150.0)

    if show_animation:  # pragma: no cover
        plt.plot(ox, oy, ".k")
        plt.plot(sx, sy, "^r")
        plt.plot(gx, gy, "^c")
        plt.grid(True)
        plt.axis("equal")
        plt.gca().xaxis.set_ticks_position('top')
        plt.xlim(0, 400)  # Set the x-axis range
        plt.ylim(150,0 )  # Set the y-axis range
```

```
    planner=VoronoiRoadMapPlanner()
    rx, ry = planner.planning(sx, sy, gx, gy, ox, oy,
                                    robot_size)
    dist = planner.cal_euclidean_dist(rx, ry)
    print("distance: ", dist)

    assert rx, 'Cannot found path'

    if show_animation:  # pragma: no cover
        plt.plot(rx, ry, "-r")
        plt.pause(0.1)
        plt.show()


if __name__ == '__main__':
    main()
```

3.2.2 Dijkstra algorithm

```
    import matplotlib.pyplot as plt
import math
import numpy as np


class DijkstraSearch:
    class Node:

        def __init__(self, x, y, cost=None, parent=None, edge_ids=None):
            self.x = x
            self.y = y
            self.cost = cost
            self.parent = parent
            self.edge_ids = edge_ids

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

    def __init__(self, show_animation):
        self.show_animation = show_animation

    def search(self, sx, sy, gx, gy, node_x, node_y, edge_ids_list):
        """
        Search shortest path

        s_x: start x positions [m]
        s_y: start y positions [m]
        gx: goal x position [m]
        gx: goal x position [m]
        node_x: node x position
        node_y: node y position
        edge_ids_list: edge_list each item includes a list of edge ids
        """

        start_node = self.Node(sx, sy, 0.0, -1)
        goal_node = self.Node(gx, gy, 0.0, -1)
        current_node = None
```

```
    open_set, close_set = dict(), dict()
    open_set[self.find_id(node_x, node_y, start_node)] = start_node

    while True:
        if self.has_node_in_set(close_set, goal_node):
            print("goal is found!")
            goal_node.parent = current_node.parent
            goal_node.cost = current_node.cost
            break
        elif not open_set:
            print("Cannot find path")
            break

        current_id = min(open_set, key=lambda o: open_set[o].cost)
        current_node = open_set[current_id]

        # show graph
        if self.show_animation and len(
                close_set.keys()) % 2 == 0:  # pragma: no cover
            plt.plot(current_node.x, current_node.y, "xg")
            # for stopping simulation with the esc key.
            plt.gcf().canvas.mpl_connect(
                'key_release_event',
                lambda event: [exit(0) if event.key == 'escape' else None])
            plt.pause(0.1)

        # Remove the item from the open set
        del open_set[current_id]
        # Add it to the closed set
        close_set[current_id] = current_node

        # expand search grid based on motion model
        for i in range(len(edge_ids_list[current_id])):
            n_id = edge_ids_list[current_id][i]
            dx = node_x[n_id] - current_node.x
            dy = node_y[n_id] - current_node.y
            d = math.hypot(dx, dy)
            node = self.Node(node_x[n_id], node_y[n_id],
                             current_node.cost + d, current_id)

            if n_id in close_set:
                continue
            # Otherwise if it is already in the open set
            if n_id in open_set:
                if open_set[n_id].cost > node.cost:
                    open_set[n_id] = node
            else:
                open_set[n_id] = node

    # generate final course
    rx, ry = self.generate_final_path(close_set, goal_node)

    return rx, ry

@staticmethod
def generate_final_path(close_set, goal_node):
    rx, ry = [goal_node.x], [goal_node.y]
```

```python
    parent = goal_node.parent
    while parent != -1:
        n = close_set[parent]
        rx.append(n.x)
        ry.append(n.y)
        parent = n.parent
    rx, ry = rx[::-1], ry[::-1]  # reverse it
    return rx, ry

def has_node_in_set(self, target_set, node):
    for key in target_set:
        if self.is_same_node(target_set[key], node):
            return True
    return False

def find_id(self, node_x_list, node_y_list, target_node):
    for i, _ in enumerate(node_x_list):
        if self.is_same_node_with_xy(node_x_list[i], node_y_list[i],
                                     target_node):
            return i
    return None

@staticmethod
def is_same_node_with_xy(node_x, node_y, node_b):
    dist = np.hypot(node_x - node_b.x,
                    node_y - node_b.y)
    return dist <= 0.1

@staticmethod
def is_same_node(node_a, node_b):
    dist = np.hypot(node_a.x - node_b.x,
                    node_a.y - node_b.y)
    return dist <= 0.1
```

3.3 Results

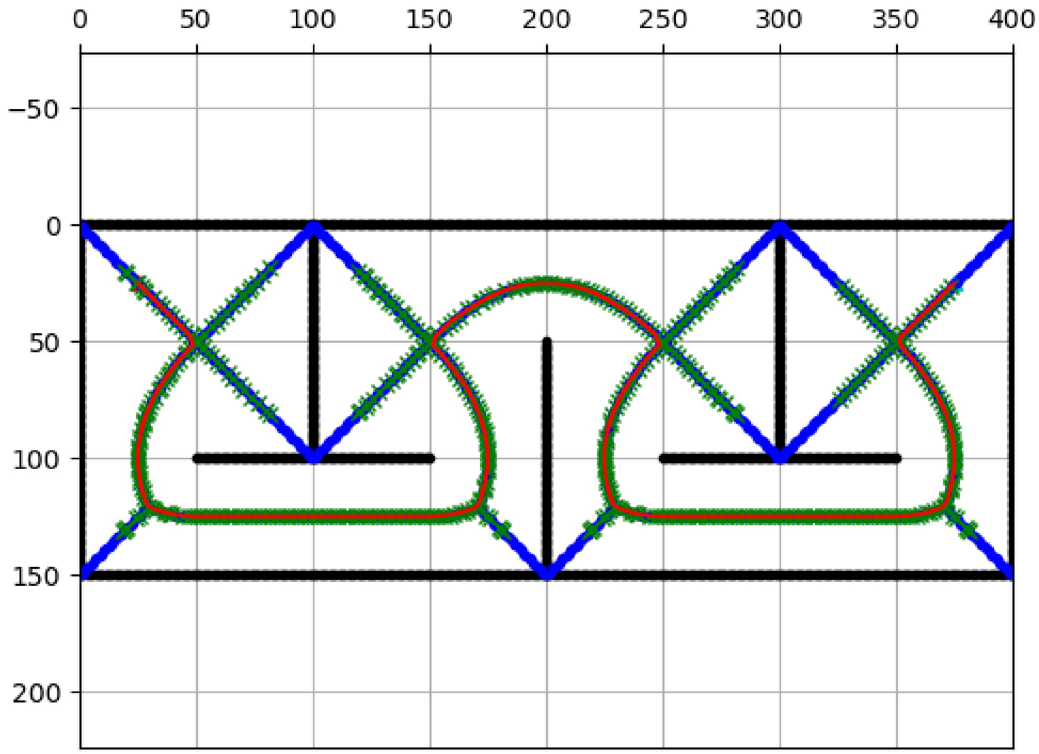


Figure 6: safest path

4 Question 4

Use probabilistic roadmaps (PRM) to compute a path between start and end-points with 50, 100 and 500 sample points. What is the difference in path length? Illustrate each computed path.

4.1 Method

The Probabilistic Road Map (PRM) is a motion planning algorithm that creates a roadmap by randomly sampling collision-free configurations in the space and connecting them with valid paths to form a graph. The construction phase generates this graph by first sampling random points and then attempting to connect each point to its k-nearest neighbors with collision-free paths, while the query phase connects start and goal positions to the roadmap and uses graph search algorithms like Dijkstra's to find a path through the network. Due to its probabilistic nature, PRM can effectively handle complex, high-dimensional configuration spaces, though the quality of the solution depends on the number of samples and the sampling strategy used. Moreover, I use the Euclidean distance formula to compute the length of the path in different sample points.I use the Euclidean distance formula to compute the length of the path for different sample sizes. Based on the results:

- With 500 sample points, the PRM algorithm traverses approximately 654 grids
- With 100 sample points,it traverses approximately 690 grids
- With 50 sample points,it traverses approximately 794 grids

4.2 Code

```
    import math
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial import KDTree

# parameter
N_SAMPLE = 500  # number of sample_points
N_KNN = 20 # number of edge from one sampled point
MAX_EDGE_LEN = 20.0  # [m] Maximum edge length

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:
    """
    obstacle_kd_tree = KDTree(np.vstack((obstacle_x_list, obstacle_y_list)).T)

    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")

    road_map = generate_road_map(sample_x, sample_y,
                                 robot_radius, obstacle_kd_tree)

    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):
    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)

    # goal point check
    dist, _ = obstacle_kd_tree.query([gx, gy])
    if dist <= rr:
        return True  # collision

    return False  # OK


def generate_road_map(sample_x, sample_y, rr, obstacle_kd_tree):
    """
    Road map generation

    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)

    #plot_road_map(road_map, sample_x, sample_y)

    return road_map


def dijkstra_planning(sx, sy, gx, gy, road_map, sample_x, sample_y):
    """
    s_x: start x position [m]
    s_y: start y position [m]
    goal_x: goal x position [m]
    goal_y: goal y position [m]
    obstacle_x_list: x position list of Obstacles [m]
    obstacle_y_list: y position list of Obstacles [m]
    robot_radius: robot radius [m]
    road_map: ??? [m]
    sample_x: ??? [m]
    sample_y: ??? [m]

    @return: Two lists of path coordinates ([x1, x2, ...], [y1, y2, ...]), empty list when no path w
    """

    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]

        # show graph
        if show_animation and len(closed_set.keys()) % 2 == 0:
            # for stopping simulation with the esc key.
            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

        # Remove the item from the open set
        del open_set[c_id]
        # Add it to the closed set
        closed_set[c_id] = current

        # expand search grid based on motion model
        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
            # Otherwise if it is already in the open set
            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 path_found is False:
        return [], []

    # generate final course
    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):  # pragma: no cover

    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 cal_euclidean_dist(rx,ry):
        dist = 0
        for i in range(len(rx)-1):
            point1 = np.array([rx[i], ry[i]])
            point2 = np.array([rx[i+1], ry[i+1]])
            dist = dist + np.linalg.norm(point1 - point2)
        return dist

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

    # start and goal position
    sx = 25.0  # [m]
    sy = 25.0  # [m]
    gx = 375 # [m]
    gy = 25.0  # [m]
    robot_size = 18
    # [m]

    ox, oy = [], []
    for i in range(50, 151):
        ox.append(i)
        oy.append(100.0)
    for i in range(0, 101):
        ox.append(100.0)
        oy.append(i)
    for i in range(50, 151):
        ox.append(200.0)
        oy.append(i)
    for i in range(250, 351):
        ox.append(i)
        oy.append(100.0)
    for i in range(0, 101):
        ox.append(300.0)
        oy.append(i)
    for i in range(0, 151):
        ox.append(0.0)
```

```
        oy.append(i)
    for i in range(0, 401):
        ox.append(i)
        oy.append(0.0)
    for i in range(0, 151):
        ox.append(400.0)
        oy.append(i)
    for i in range(0, 401):
        ox.append(i)
        oy.append(150.0)

    if show_animation:
        #plt.plot(cspace_x, cspace_y, ".", color = 'gray')
        plt.plot(ox, oy, ".k")
        plt.plot(sx, sy, "^r")
        plt.plot(gx, gy, "^c")
        plt.gca().invert_yaxis()
        plt.grid(True)
        plt.axis("equal")

    rx, ry = prm_planning(sx, sy, gx, gy, ox, oy, robot_size, rng=rng)
    dist =cal_euclidean_dist(rx, ry)
    print("distance: ", dist)
    if show_animation:
        plt.plot(rx, ry, "-r")
        plt.pause(0.001)
        plt.show()


if __name__ == '__main__':
    main()
```

4.3 Results

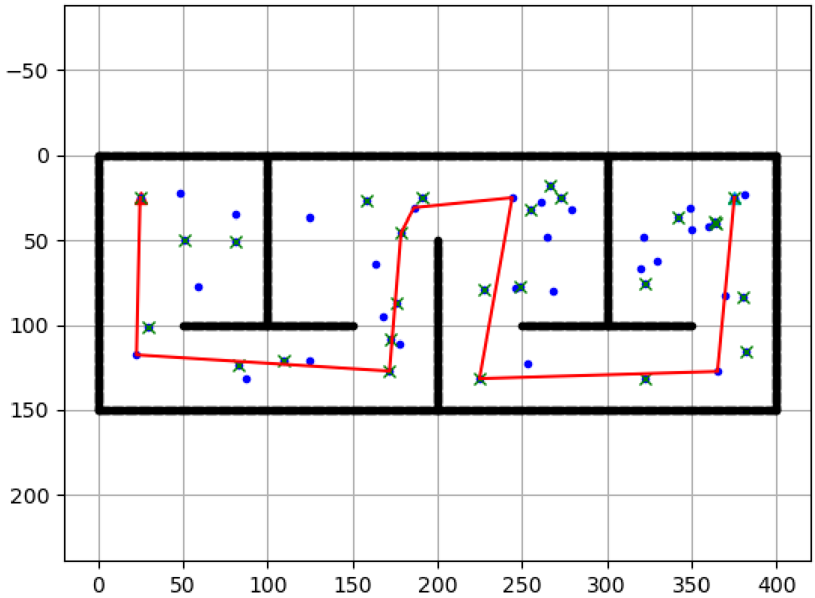


Figure 7: PRM with sample points 50

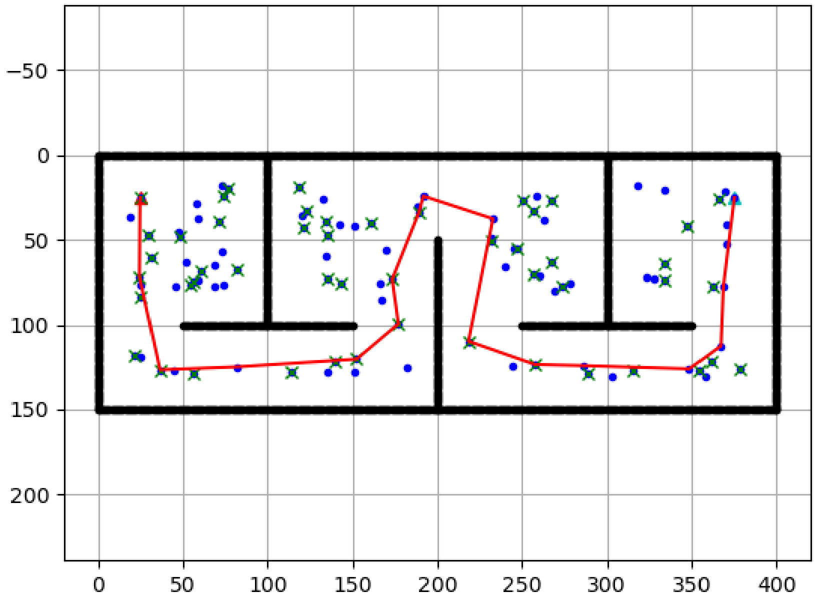


Figure 8: PRM with sample points 100

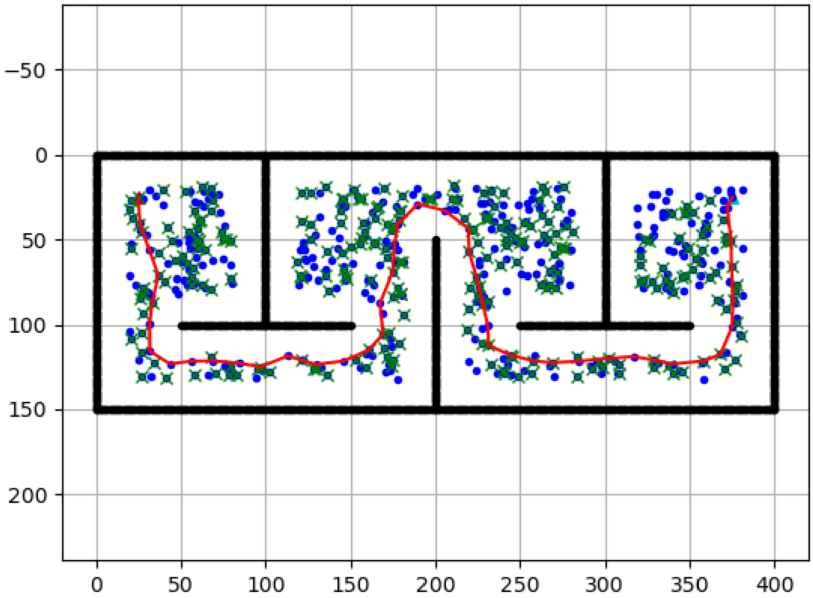


Figure 9: PRM with sample points 500

5 Question 5

Do the same with Rapid exploring random trees (RRT). What are the main differences in performance between PRM and RRT? Illustrate each path.

5.1 Method

Rapidly-exploring Random Tree (RRT) is a sampling-based algorithm that builds a space-filling tree incrementally from randomly generated points, starting from the initial position and growing towards the unexplored regions until it reaches the goal. Unlike PRM which creates a complete roadmap of connections between random points in advance (learning phase) and then searches for a path (query phase), RRT builds and explores paths simultaneously by growing a tree structure directly from the start position, making it more efficient for single-query problems. The main difference is that PRM is better for multiple queries in the same environment since you can reuse the roadmap, while RRT is more efficient for single-query problems and can handle non-holonomic constraints better since it builds paths incrementally in the direction of motion. I have experimented with different maximum edge lengths in the PRM algorithm, and the results are as follows:

- With a maximum edge length of 50, the path length is approximately 854.6 grids.
- With a maximum edge length of 75, the path length is approximately 783 grids.
- With a maximum edge length of 100, the path length is approximately 775 grids.

5.2 Code

```
    import math
import random
import time
import matplotlib.pyplot as plt
import numpy as np
show_animation = True
```

```
class RRT:
    """
    Class for RRT planning
    """

    class Node:
        """
        RRT Node
        """

        def __init__(self, x, y):
            self.x = x
            self.y = y
            self.path_x = []
            self.path_y = []
            self.parent = None

    class AreaBounds:

        def __init__(self, area):
            self.xmin = float(area[0])
            self.xmax = float(area[1])
            self.ymin = float(area[2])
            self.ymax = float(area[3])


    def __init__(self,
                 start,
                 goal,
                 obstacle_list,
                 rand_area,
                 expand_dis=75.0,
                 path_resolution=5,
                 goal_sample_rate=5,
                 max_iter=5000,
                 play_area=None,
                 robot_radius=0.0,
                 ):
        """
        Setting Parameter

        start:Start Position [x,y]
        goal:Goal Position [x,y]
        obstacleList:obstacle Positions [[x,y,size],...]
        randArea:Random Sampling Area [min,max]
        play_area:stay inside this area [xmin,xmax,ymin,ymax]
        robot_radius: robot body modeled as circle with given radius

        """
        self.start = self.Node(start[0], start[1])
        self.end = self.Node(goal[0], goal[1])
        # minimum and maximum random sampling area
        self.min_rand_x = rand_area[0]
        self.max_rand_x = rand_area[1]
        self.min_rand_y = rand_area[2]
        self.max_rand_y = rand_area[3]
        if play_area is not None:
```

```
            self.play_area = self.AreaBounds(play_area)
        else:
            self.play_area = None
        self.expand_dis = expand_dis
        self.path_resolution = path_resolution
        self.goal_sample_rate = goal_sample_rate
        self.max_iter = max_iter
        self.obstacle_list = obstacle_list
        self.node_list = []
        self.robot_radius = robot_radius

    def planning(self, animation=None):
        """
        rrt path planning

        animation: flag for animation on or off
        """
        self.node_list = [self.start]
        for i in range(self.max_iter):
            rnd_node = self.get_random_node()
            nearest_ind = self.get_nearest_node_index(self.node_list, rnd_node)
            nearest_node = self.node_list[nearest_ind]

            new_node = self.steer(nearest_node, rnd_node, self.expand_dis)

            if self.check_if_outside_play_area(new_node, self.play_area) and \
               self.check_collision(
                   new_node, self.obstacle_list, self.robot_radius):
                self.node_list.append(new_node)

            if animation and i % 50 == 0:
                self.draw_graph(rnd_node)

            if self.calc_dist_to_goal(self.node_list[-1].x,
                                      self.node_list[-1].y) <= self.expand_dis:
                final_node = self.steer(self.node_list[-1], self.end,
                                        self.expand_dis)
                if self.check_collision(
                        final_node, self.obstacle_list, self.robot_radius):
                    return self.generate_final_course(len(self.node_list) - 1)

            if animation and i % 50:
                self.draw_graph(rnd_node)

        return None  # cannot find path


    def steer(self, from_node, to_node, extend_length=float("inf")):

        new_node = self.Node(from_node.x, from_node.y)
        d, theta = self.calc_distance_and_angle(new_node, to_node)

        new_node.path_x = [new_node.x]
        new_node.path_y = [new_node.y]

        if extend_length > d:
            extend_length = d
```

```
        n_expand = math.floor(extend_length / self.path_resolution)

        for _ in range(n_expand):
            new_node.x += self.path_resolution * math.cos(theta)
            new_node.y += self.path_resolution * math.sin(theta)
            new_node.path_x.append(new_node.x)
            new_node.path_y.append(new_node.y)

        d, _ = self.calc_distance_and_angle(new_node, to_node)
        if d <= self.path_resolution:
            new_node.path_x.append(to_node.x)
            new_node.path_y.append(to_node.y)
            new_node.x = to_node.x
            new_node.y = to_node.y

        new_node.parent = from_node

        return new_node

    def generate_final_course(self, goal_ind):
        path = [[self.end.x, self.end.y]]
        node = self.node_list[goal_ind]
        while node.parent is not None:
            path.append([node.x, node.y])
            node = node.parent
        path.append([node.x, node.y])
        print("length of node_list",len(self.node_list))
        return path

    def calc_dist_to_goal(self, x, y):
        dx = x - self.end.x
        dy = y - self.end.y
        return math.hypot(dx, dy)

    def cal_euclidean_dist(self,rx,ry):
        dist = 0
        for i in range(len(rx)-1):
            point1 = np.array([rx[i], ry[i]])
            point2 = np.array([rx[i+1], ry[i+1]])
            dist = dist + np.linalg.norm(point1 - point2)
        return dist

    def get_random_node(self):
        if random.randint(0, 100) > self.goal_sample_rate:
            rnd = self.Node(
                random.uniform(self.min_rand_x, self.max_rand_x),
                random.uniform(self.min_rand_y, self.max_rand_y))
        else:  # goal point sampling
            rnd = self.Node(self.end.x, self.end.y)
        return rnd

    def draw_graph(self, rnd=None):
        plt.clf()
        # for stopping simulation with the esc key.
        plt.gcf().canvas.mpl_connect(
            'key_release_event',
            lambda event: [exit(0) if event.key == 'escape' else None])
        if rnd is not None:
```

```
        plt.plot(rnd.x, rnd.y, "^k")
        if self.robot_radius > 0.0:
            self.plot_circle(rnd.x, rnd.y, self.robot_radius, '-r')
    #plot tree
    for node in self.node_list:
        if node.parent:
            plt.plot(node.path_x, node.path_y, "-g")

    obstacleList = generate_obstacle_list()


    for (ox, oy) in self.obstacle_list:
        self.plot_line(ox, oy)
    #plot play area
    #if self.play_area is not None:
        # no plot for now
        # plt.plot([self.play_area.xmin, self.play_area.xmax,
        #           self.play_area.xmax, self.play_area.xmin,
        #           self.play_area.xmin],
        #          [self.play_area.ymin, self.play_area.ymin,
        #           self.play_area.ymax, self.play_area.ymax,
        #           self.play_area.ymin],
        #          "-k")

    plt.plot(self.start.x, self.start.y, "xr")
    plt.plot(self.end.x, self.end.y, "xr")
    plt.axis("equal")
    plt.axis([0, 401, 0, 151])
    plt.gca().invert_yaxis()
    plt.grid(True)
    plt.pause(0.0001)

@staticmethod
def plot_circle(x, y, size, color="-b"):  # pragma: no cover
    deg = list(range(0, 360, 5))
    deg.append(0)
    xl = [x + size * math.cos(np.deg2rad(d)) for d in deg]
    yl = [y + size * math.sin(np.deg2rad(d)) for d in deg]
    plt.plot(xl, yl,".",color="gray")

@staticmethod
def plot_line(x, y, color="black"):  # pragma: no cover
    plt.plot(x, y,".",color=color)

@staticmethod
def get_nearest_node_index(node_list, rnd_node):
    dlist = [(node.x - rnd_node.x)**2 + (node.y - rnd_node.y)**2
             for node in node_list]
    minind = dlist.index(min(dlist))

    return minind

@staticmethod
def check_if_outside_play_area(node, play_area):

    if play_area is None:
        return True  # no play_area was defined, every pos should be ok
```

```
        if node.x < play_area.xmin or node.x > play_area.xmax or \
           node.y < play_area.ymin or node.y > play_area.ymax:
            return False  # outside - bad
        else:
            return True  # inside - ok

    @staticmethod
    def check_collision(node, obstacleList, robot_radius):

        if node is None:
            return False

        for (ox, oy) in obstacleList:
            dx_list = [ox - x for x in node.path_x]
            dy_list = [oy - y for y in node.path_y]
            d_list = [dx * dx + dy * dy for (dx, dy) in zip(dx_list, dy_list)]

            if min(d_list) <= (robot_radius)**2:
                return False  # collision

        return True  # safe

    @staticmethod
    def calc_distance_and_angle(from_node, to_node):
        dx = to_node.x - from_node.x
        dy = to_node.y - from_node.y
        d = math.hypot(dx, dy)
        theta = math.atan2(dy, dx)
        return d, theta

def generate_obstacle_list():
    obstacle_list = []

    for i in range(0, 400):
        obstacle_list.append((i, 0.0))
        # ox.append(i)
        # oy.append(0.0)

    for i in range(0, 150):
        obstacle_list.append((400.0, i))
        # ox.append(400.0)
        # oy.append(i)

    for i in range(0, 401):
        obstacle_list.append((i, 150.0))
        # ox.append(i)
        # oy.append(150.0)

    for i in range(0, 151):
        obstacle_list.append((0.0, i))
        # ox.append(0.0)
        # oy.append(i)

    #obstacles
    #left obstacle
    for i in range(0, 100):
        obstacle_list.append((99.0, i))
        # ox.append(99.0)
```

```
        # oy.append(i)

    for i in range(0, 100):
        obstacle_list.append((50.0+i, 99.0))
        # ox.append(50.0 + i)
        # oy.append(99.0)

        #mid obstacles
    for i in range(0, 100):
        obstacle_list.append((200.0, 50.0+i))
        # ox.append(200.0)
        # oy.append(50.0 + i)

    #right obstacle
    for i in range(0, 100):
        obstacle_list.append((299.0, i))
        # ox.append(299.0)
        # oy.append(i)

    for i in range(0, 100):
        obstacle_list.append((250.0+i, 99.0))
        # ox.append(250.0 + i)
        # oy.append(99.0)

    return obstacle_list

def main(gx=750.0/2, gy=25.0):
    print("start " + __file__)

    # ====Search Path with RRT====
    obstacleList = generate_obstacle_list()
    # obstacleList = [(5, 5, 1), (3, 6, 2), (3, 8, 2), (3, 10, 2), (7, 5, 2),
    #                 (9, 5, 2), (8, 10, 1)]  # [x, y, radius]
    # Set Initial parameters
    rrt = RRT(
        start=[25, 25],
        goal=[gx, gy],
        rand_area=[18, 400-18, 18, 150-18],
        obstacle_list=obstacleList,
        play_area=[18,400-18,18, 150-18],
        robot_radius=16#12.5*math.sqrt(2)
        )
    path = rrt.planning(animation=show_animation)

    if path is None:
        print("Cannot find path")
    else:
        print("found path!!")
        rx = [x for (x, y) in path]
        ry = [y for (x, y) in path]
        dist = rrt.cal_euclidean_dist(rx, ry)
        print("distance: ", dist)
        # Draw final path
        if show_animation:
            rrt.draw_graph()
            plt.plot([x for (x, y) in path], [y for (x, y) in path], '-r')
            plt.grid(True)
            plt.pause(0.0001)  # Need for Mac
```

```
            plt.show()


if __name__ == '__main__':
    main()
```

5.3 Results

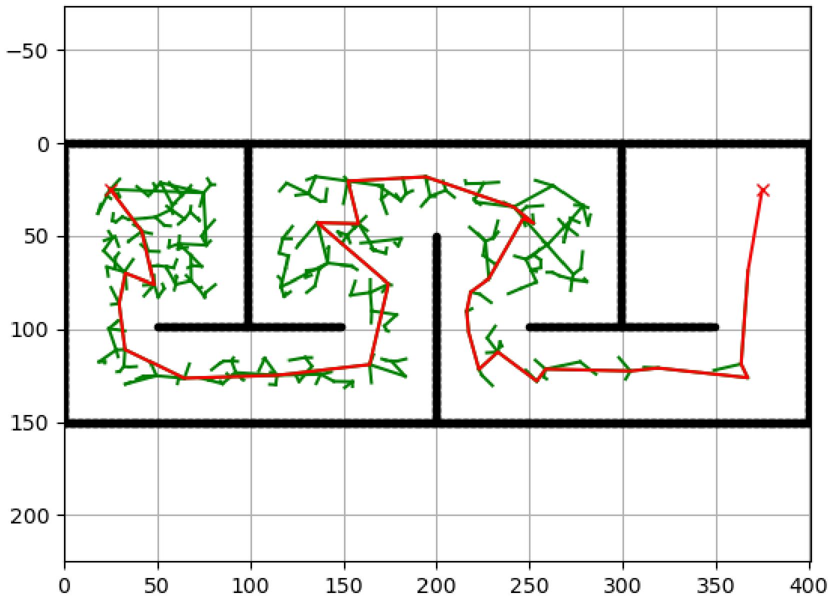


Figure 10: RRT with maximum edge length(50)

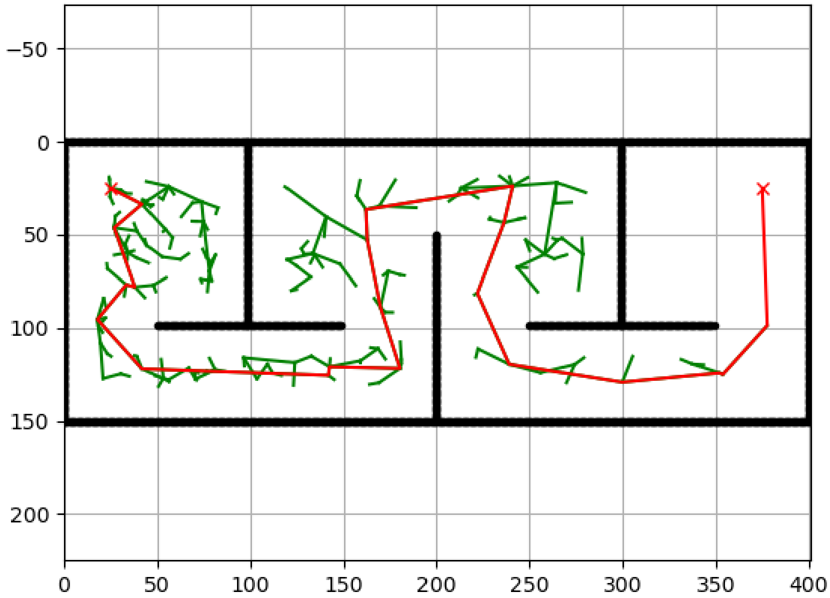


Figure 12: RRT with maximum edge length(100)

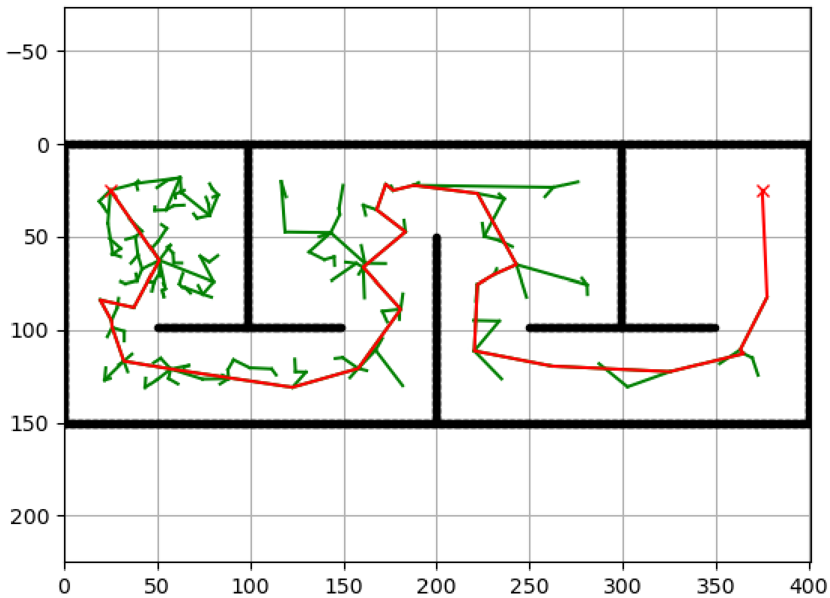


Figure 11: RRT with maximum edge length(75)