CSE276C HW5

Haakon Kjelås Garfjell

December 2023

1 Configuration space

Generate the configuration space for the robot with a grid size of 2x2 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.

In order to generate the configuration space with a grid size of 2x2, I define the following parameters:

$$\text{robotWidth} = 50/2 = 25$$
$$\text{robotHeight} = 50/2 = 25$$
$$\text{spaceWidth} = 800/2 = 400$$
$$\text{spaceHeight} = 300/2 = 150$$

I then create the work space as a numpy array of size spaceHeight x spaceWidth, where elements of value 0 represent free cells and elements of value 1 represent occupied cells. This is shown in figure 1.

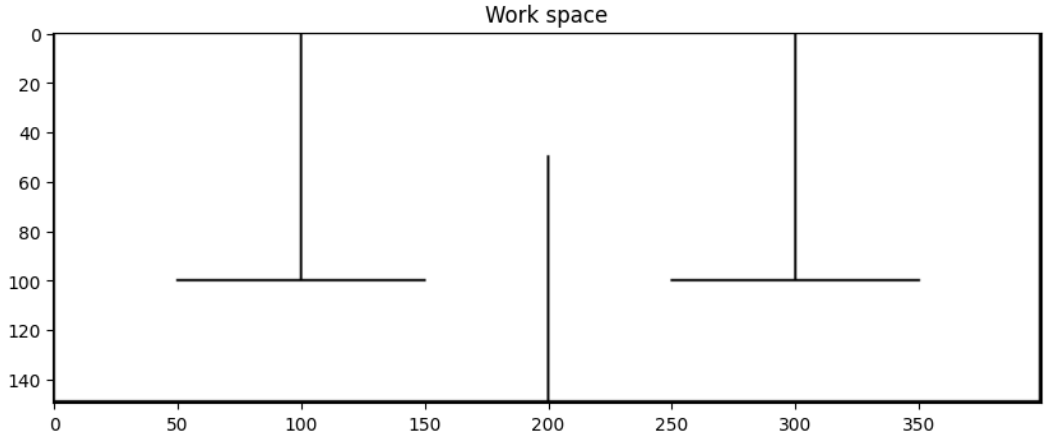

Figure 1: Orginal work space reduced to a 2x2 grid size

To create the configuration space I create a binary mask representing the robot at angles 0 to 180 degrees with a resolution of 5 degrees. I then use convolve2d from scipy.signal to apply the convolution to the original workspace. I stack each of the convoluted spaces on top of each other to create a 3D configuration space where each plane (x,y) represents the configuration space for a given angle (z).

The configuration spaces for 0, 45 and 90 degrees can be seen in figure 2.

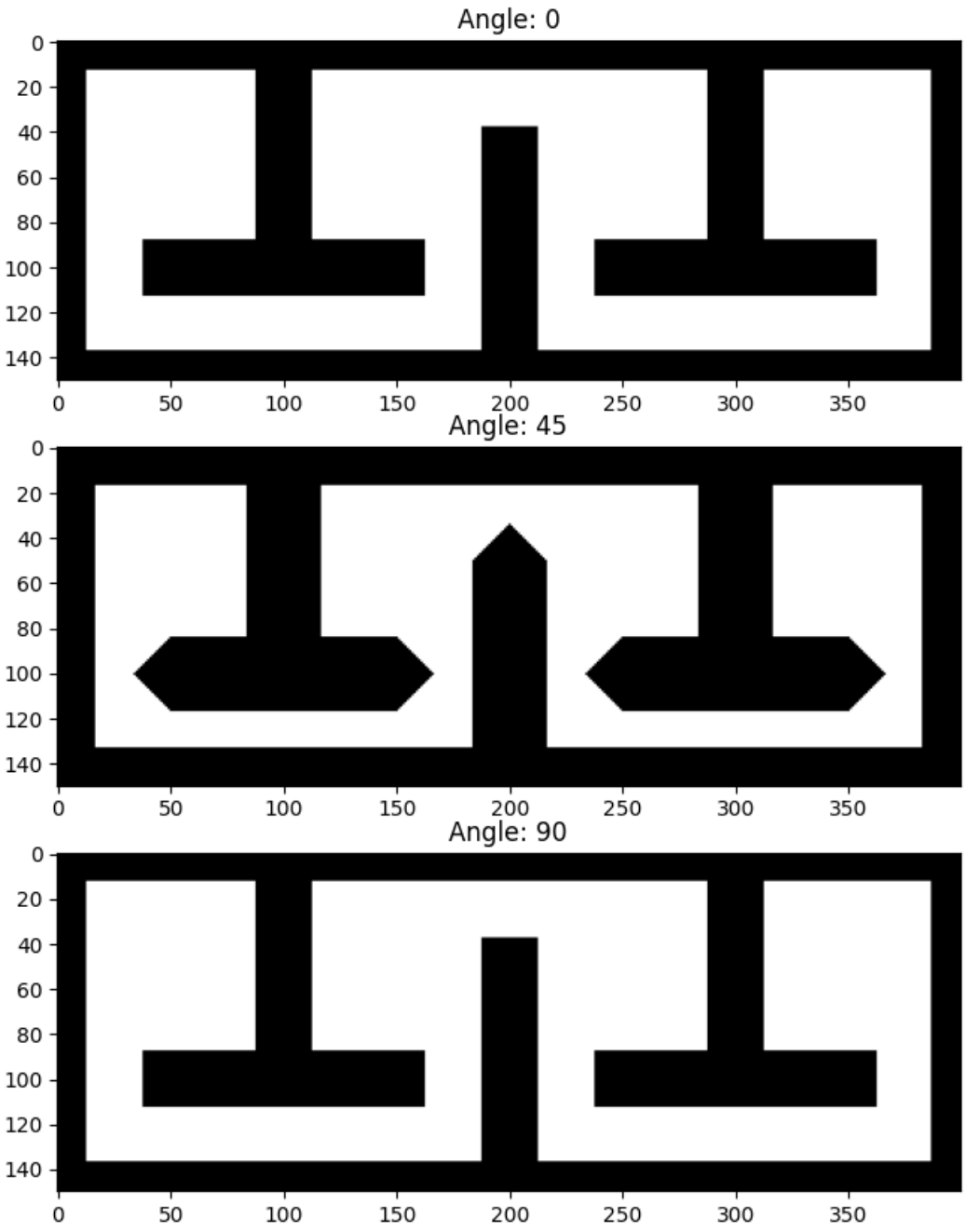

Figure 2: Configuration space for angles 0, 45 and 90

2 Greedy search

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.

To find the shortest path I implemented the A* algorithm as it combines Greedy best first and Dijkstra. This makes it find an optimal path while at the same time prioritizing the nodes with the lowest total cost.

The total cost is defined as:

$$f(n) = g(n) + h(n)$$

I define the g-cost to increase by 1 for each tile in the path, while the h-cost is the euclidean distance to the target node. In my implementation I let the g-cost also increase for rotation, however it can be argued only movement in the plane should be considered.

For my implementation I let all directions be valid, this means the robot is allowed to both move and rotate in all directions. For example moving diagonally and rotating 5 degrees at the same time is a valid direction.
My algorithm can be broken down into 5 steps:

1. Initialize priority queue with the source node, a cost of 0 and an empty path. I use heapq for this.
2. Pop node from priority queue.
    1. If node is target, terminate and return path.
    2. If node is already visited, continue.
3. Add node to visited.
4. Loop through neighbors and calculate total cost (f(n)).
5. Push node with cost and path to node onto the priority queue.

In this case a node is the same thing a tile in the 3D configuration space.

A 2D projection of the path can be seen in figure 3. Although it looks like the robot violates the constraints of the obstacles in the corners of the walls, it is a result of the robot having a different orientation. The actual path it takes can be seen in figure 4.

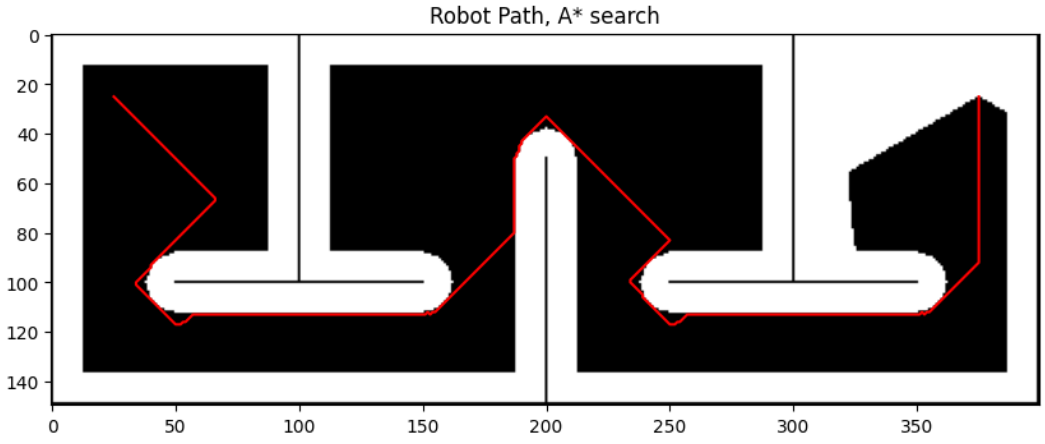

Figure 3: A* search. Black tiles that are not part of the walls are the 2D projection of the visited nodes.

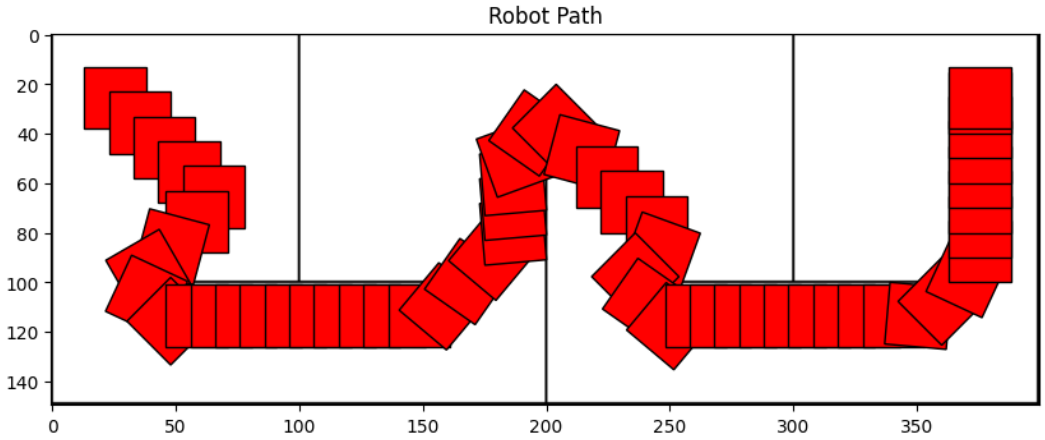

Figure 4: A* search. Black tiles that are not part of the walls are the 2D projection of the visited nodes.

This gave a path length of 552. I calculate the path length by only considering movement in the plane, meaning a single rotation does not add to the length (adding rotations to the path length just increased it by one to 553 in this case).

3 Safest path

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

For this task I experimented with medial_axis from scipy.morphology, however I could not find support for 3D grids (which we need to take all the configuration spaces into account). I ended up using skeletonize_3d, also from scipy.morphology. It works by making successive passes of the image. On each pass, border pixels are identified and removed on the condition that they do not break the connectivity of the corresponding object [1].

The resulting skeleton of the empty cells gives the path that is furthest away from the walls at all times. This can be seen in figure 5. However, as seen from the figure, the safe path is not connected to the start and the endpoint. In order to find the complete path, I first identify which node in the safe path is closest in euclidean distance to the start node. I then create an empty field, and apply A* to find the path form the start node to the closest node in the path. I repeat this for the target node. The complete safe path can be seen in 6.

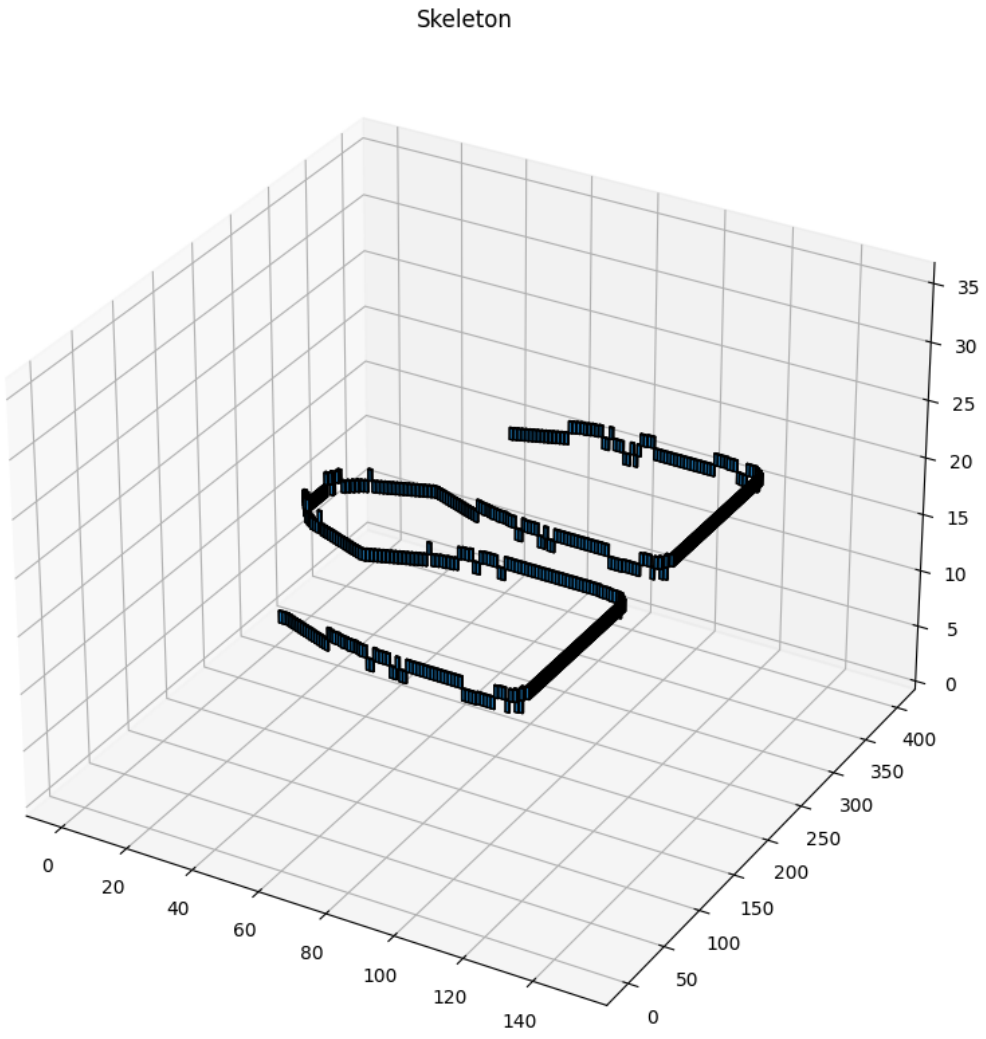

Figure 5: Skeleton of 3D configuration space

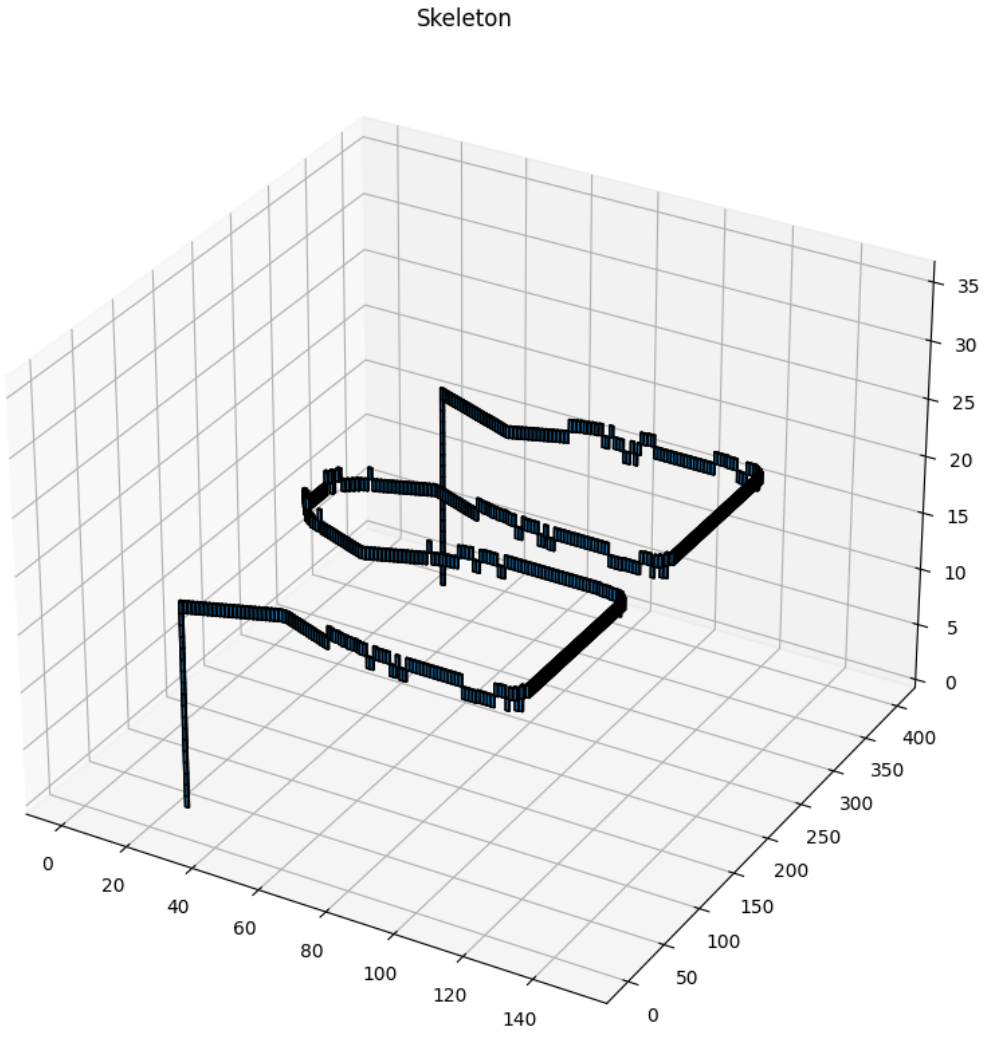

Figure 6: Skeleton represents safe path in 3D

What the path looks like in the original work space can be seen in figure 7. We can clearly see that we at all times try to stay as far away from the walls as possible, even though it means our path length is longer.

We can also see from 6 that there is little to no rotation of the robot. It mostly just rotates to get from the start orientation of 0 degrees to around 15 degrees. This can also be seen from the robot trajectory on the path shown in figure 8. In this example we could have gotten just as good a result by just using a single configuration space, but if the obstacles demand more precise movement of the robot (for example if it has to rotate to fit into a small space) we could still find a good path.

The length of the safest path by the same computation as in the previous task is 727 (considering rotations it is 760). It makes sense that this path is longer than in the A* case as we no longer care about finding the shortest path.

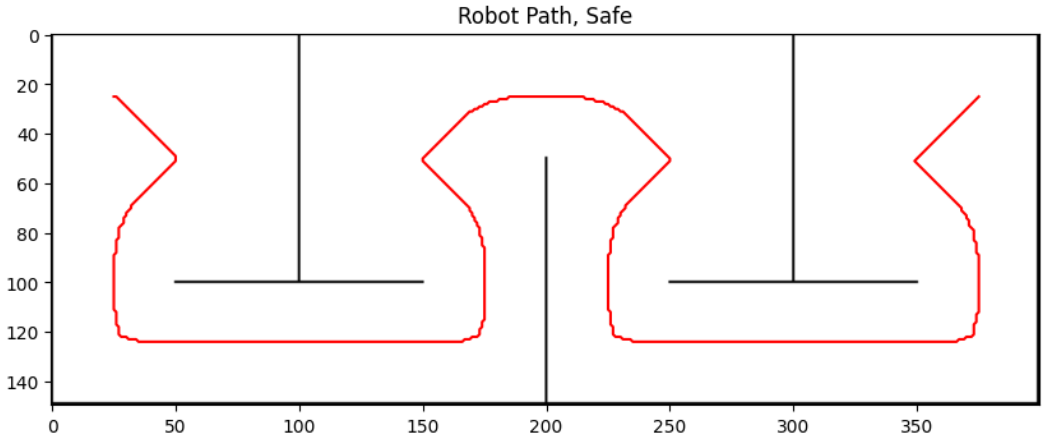

Figure 7: Projection of the safest path in original work space

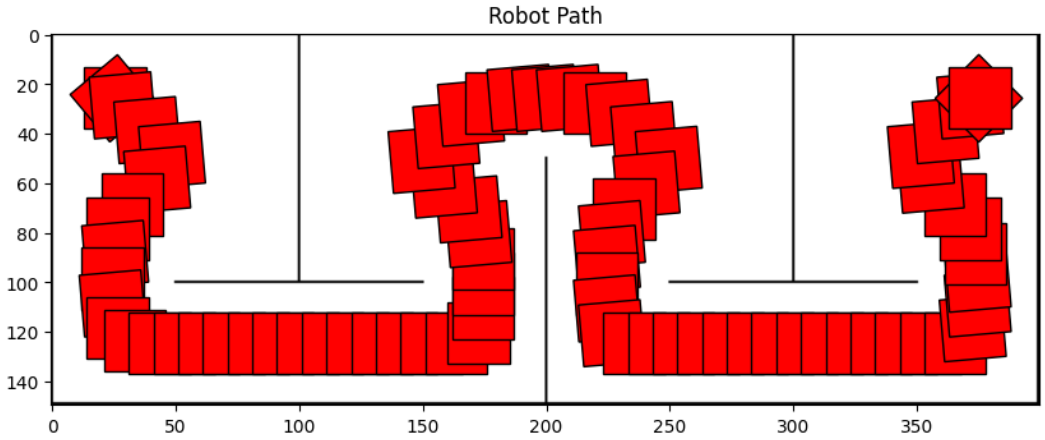

Figure 8: Projection of the safest path in original work space

4 PRM

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.

For this task I am using the networkx library to create graphs that can later be used to find the shortest path between the source and target. The implementation also uses the 2D configuration space where the angle is 0 degrees to better illustrate how the algorithm behaves.

To create a probabilistic roadmap I can divide my algorithm into three steps:

1. Generate n random points and include the source and target point. If a generated point is not a free cell, generate a new one.
2. For each generated point, connect the edges if there is a straight, collision free line between them.
3. Apply a path finding algorithm to the resulting graph, to find a path between source and target. I used Dijkstra.

It is worth noting that I am weighting the edges with the euclidean distance between the two points meaning that the path finding algorithm considers the actual distance when computing the shortest path and not the number of nodes required.

The path for 50, 100 and 500 sampled points can be seen in figures 9, 10 and 11. Although the graphs are plotted in the original workspace, the configuration space was used to compute them, this can be seen in figure 11.

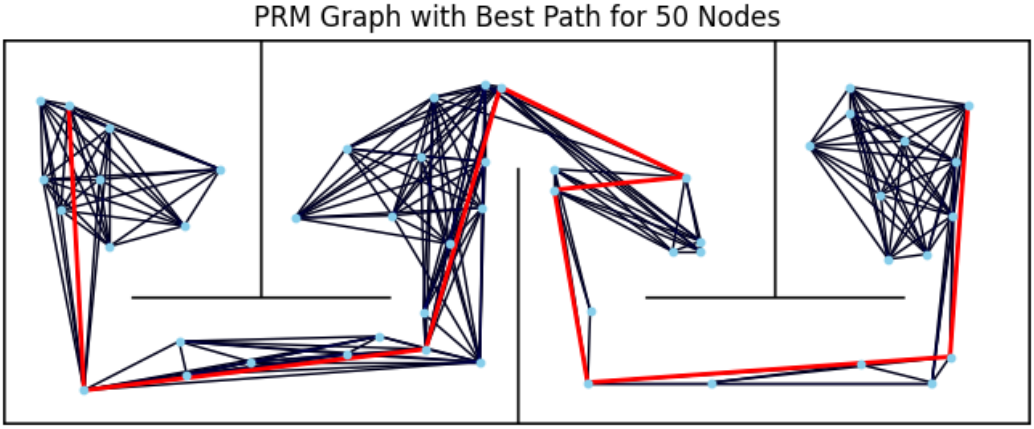

Figure 9: Shortest path in probabilistic roadmap, 50 samples

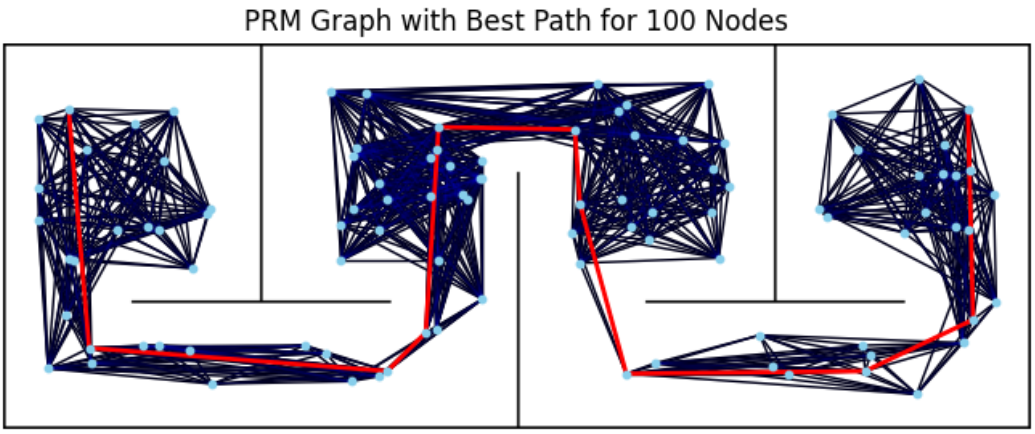

Figure 10: Shortest path in probabilistic roadmap, 100 samples

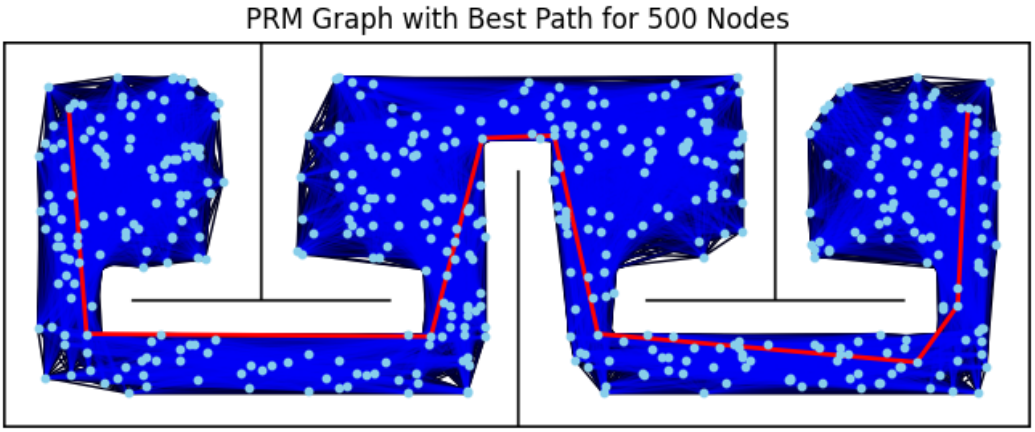

Figure 11: Shortest path in probabilistic roadmap, 500 samples

The biggest downside to using probabilistic roadmaps is that there will not always be a connected path that does not violate the constraints of the obstacles. This is especially the case for 50 samples, where I had to run the algorithm a couple of times before I was able to find a path.

An advantage is however that it reduces the configuration space down to much fewer nodes than using the entire grid, thus making path finding much more efficient.

The path lengths of the shown paths are 784, 670 and 628 for 50, 100 and 500 samples respectively. The general trend seems to be that increasing the number of samples makes for a better path, but since the points are chosen at random I observed that the 50 sample case sometimes outperformed the 100 sample, but the 500 sample case was almost always better than the two others.

5 RRT

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

As for the PRM case i used the networkx library to create the graphs, and find the shortest path between the source and target. Here I also applied the algorithm to the configuration space where the robot angle is 0 degrees.

To apply rapid exploring trees I implemented a seven step algorithm based on the Robot planning slides:

1. Add source to graph.
2. Generate random point q_rand in configuration space.
3. Find the node q_near in graph closest to the generated point.
4. Create a new node q_new, with a distance Δq from q_near towards q_rand.
5. Connect q_new to q_near if there is a collision free line between q_new and q_near. If not go to step 2.
6. If the distance from q_new to target is less than Δq, connect target to q_new.
7. traverse the tree from target back to source to find the path. (I just applied A* for convenience).

The graphs for a maximum of 50, 100, and 500 nodes and Δq = 20 can be seen in 12, 13 and 14 respectively. As for the PRM case, the graphs are plotted in the orginal work space while the algorithm has been applied to the configuration space.

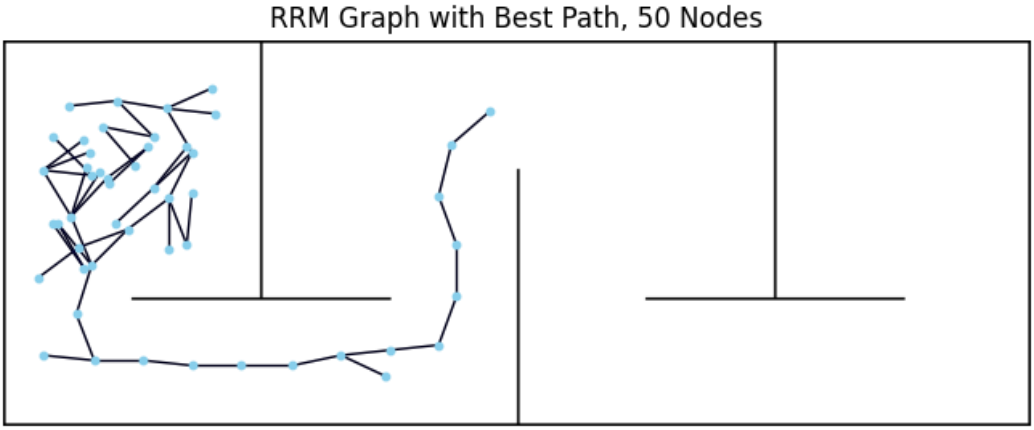

Figure 12: RRT 50 nodes

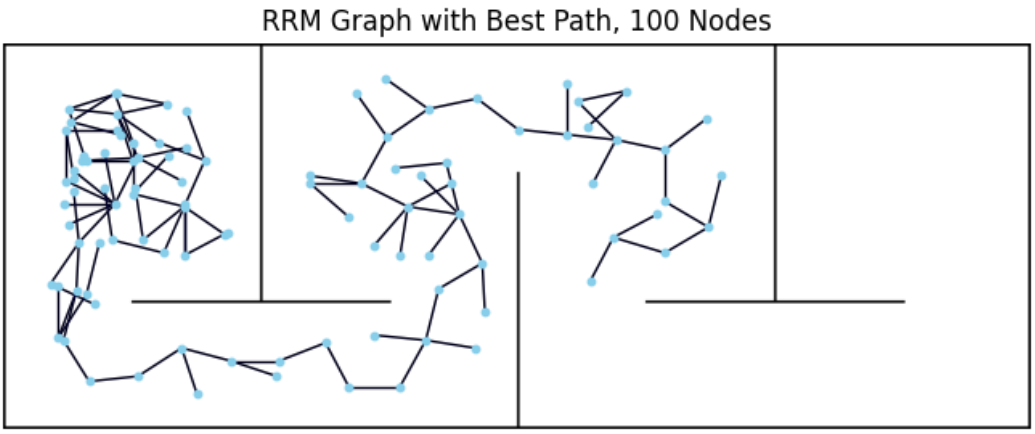

Figure 13: RRT 100 nodes

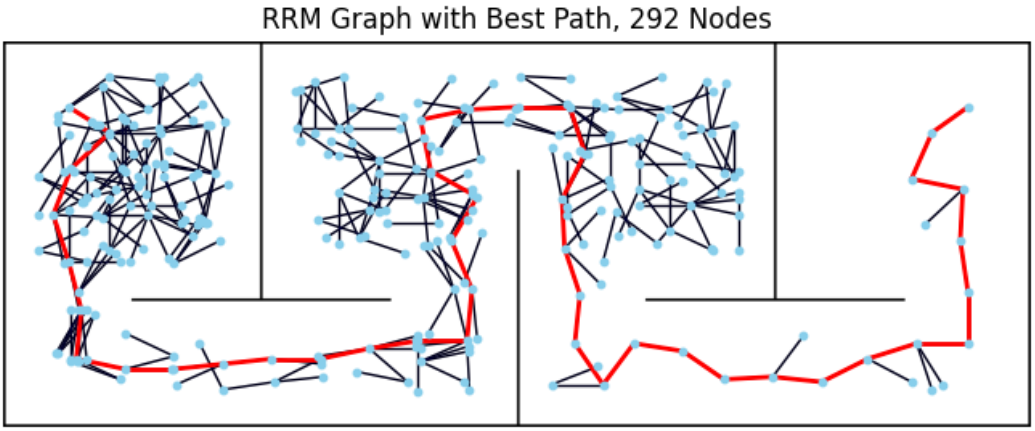

Figure 14: RRT 500 nodes

As we can see from the figures, the tree does not grow large enough for the 50 and 100 node case where we are able to find a path from the source to the target. However for the 500 node case we can terminate earlier as the tree is able to connect to the target node before we have added 500 nodes. This gives a path length of 776 which is a bit better than the 50 sample case in the PRM algorithm.

There performance of the algorithm depends on how you tune Δq. If I let $\Delta q = 45$ I am able to find a path with just 83 nodes as illustrated in figure 15. It also actually beat the 500 node limit case with a path length of 765. However, this is not consistent, and more times than not the tree is not able to connect to the source node.

Another way to improve the algorithm is increasing the threshold for connecting to the target node, however this needs to be done carefully as it might lead to violations of the obstacle constraints. It could also be improved by letting the tree grow both from the source and the target node.

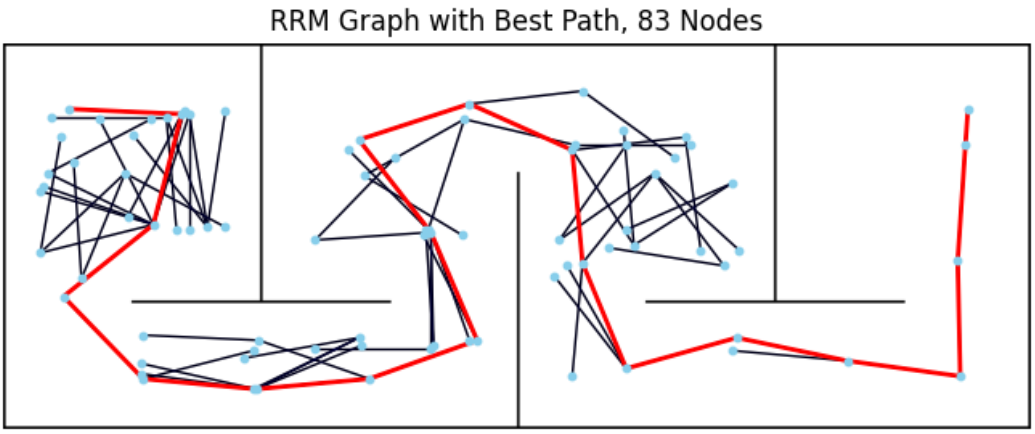

Figure 15: RRT 83 nodes

References

[1] scikit image. Skeletonize. [Online]. Available: https://scikit-image.org/docs/stable/auto/edges/plot.html

Appendix

A Code for Question 1

```
# Robot and space parameters
grid_size = 2

robot_width = 50 // grid_size
robot_height = 50 // grid_size
space_width = 800 // grid_size
space_height = 300 // grid_size

# Configuration space initialization
work_space = np.zeros((space_height, space_width), dtype=int)

# Outer walls [xmin, xmax, ymin, ymax]
outer_wall_top = np.array([[0, space_width, 0, 1]])
outer_wall_bottom = np.array([[0, space_width, space_height-1, space_height]])
outer_wall_left = np.array([[0, 1, 0, space_height]])
outer_wall_right = np.array([[space_width-1, space_width, 0, space_height]])

outer_walls = np.array([outer_wall_top, outer_wall_bottom, outer_wall_left,
    outer_wall_right])

# Inner walls [xmin, xmax, ymin, ymax]
inner_wall_1 = np.array([[200 // grid_size, 200 // grid_size + 1, 0, 200 //
    grid_size]])
inner_wall_2 = np.array([[100 // grid_size, 300 // grid_size + 1, 200 //
    grid_size, 200 // grid_size + 1]])
inner_wall_3 = np.array([[400 // grid_size , 400 // grid_size + 1, 100 //
    grid_size, 300 // grid_size + 1]])
inner_wall_4 = np.array([[600 // grid_size, 600 // grid_size + 1, 0, 200 //
    grid_size]])
inner_wall_5 = np.array([[500 // grid_size, 700 // grid_size + 1, 200 //
    grid_size, 200 // grid_size + 1]])

inner_walls = np.array([inner_wall_1, inner_wall_2, inner_wall_3, inner_wall_4,
    inner_wall_5])


for wall in outer_walls:
    work_space[wall[0, 2]:wall[0, 3], wall[0, 0]:wall[0, 1]] = 1

for wall in inner_walls:
    work_space[wall[0, 2]:wall[0, 3], wall[0, 0]:wall[0, 1]] = 1


cmap = plt.cm.gray
inverse_gray_cmap = LinearSegmentedColormap.from_list('inverse_gray', cmap(np.
    linspace(1, 0, cmap.N)))
plt.figure(figsize=(10, 10))
plt.imshow(work_space, cmap=inverse_gray_cmap)
plt.title('Work space')
```

```
plt.show()

robot_mask = np.ones((robot_height, robot_width), dtype=int)

angles = np.arange(0, 180, 5)

config_space = np.zeros((space_height, space_width, len(angles)), dtype=int)

idx_for_plot = []
for idx, angle in enumerate(angles):
    rotated_robot_mask = rotate(robot_mask, angle, order=0, reshape=True)
    rotated_robot_mask[rotated_robot_mask > 0.5] = 1
    convolved_result = convolve2d(work_space, rotated_robot_mask, mode='same',
       boundary='fill', fillvalue=0)
    convolved_result[convolved_result != 0] = 1
    config_space[:, :, idx] = convolved_result

    if angle == 0 or angle == 45 or angle == 90:
        idx_for_plot.append(idx)

cmap = plt.cm.gray
inverse_gray_cmap = LinearSegmentedColormap.from_list('inverse_gray', cmap(np.
   linspace(1, 0, cmap.N)))
plt.figure(figsize=(10, 10))
for idx in idx_for_plot:
    plt.subplot(3, 1, idx_for_plot.index(idx) + 1)
    plt.imshow(config_space[:, :, idx], cmap=inverse_gray_cmap)
    plt.title('Angle: {}'.format(angles[idx]))
plt.show()
```

B Code for Question 2

```
def astar_3d(field, start, target):
    directions = [(0, 0, 1), (0, 0, -1), (0, 1, 0), (0, -1, 0), (1, 0, 0), (-1,
       0, 0),
                  (1, 1, 0), (1, -1, 0), (-1, 1, 0), (-1, -1, 0), (1, 0, 1), (1,
                      0, -1),
                  (-1, 0, 1), (-1, 0, -1), (0, 1, 1), (0, 1, -1), (0, -1, 1),
                     (0, -1, -1),
                  (1, 1, 1), (1, 1, -1), (-1, 1, 1), (-1, 1, -1), (1, -1, 1),
                     (1, -1, -1),
                  (-1, -1, 1), (-1, -1, -1)]

    def is_valid_move(x, y, z):
        return 0 <= x < field.shape[0] and 0 <= y < field.shape[1] and 0 <= z <
           field.shape[2] and field[x, y, z] == 0

    def neighbors(x, y, z):
        valid_neighbors = []
        for dx, dy, dz in directions:
            new_x, new_y, new_z = x + dx, y + dy, z + dz
            if is_valid_move(new_x, new_y, new_z):
```

```
                valid_neighbors.append((new_x, new_y, new_z))
        return valid_neighbors

    def heuristic(a, b):
        return np.linalg.norm(np.array(a) - np.array(b))

    heap = [(0, start, [])]
    visited = set()

    while heap:
        cost, current, path = heapq.heappop(heap)

        if current == target:
            return path + [current], visited

        if current in visited:
            continue

        visited.add(current)

        for neighbor in neighbors(*current):
            new_cost = cost + 1
            priority = new_cost + heuristic(target, neighbor)
            heapq.heappush(heap, (priority, neighbor, path + [current]))

    return None

source = (50 // grid_size, 50 // grid_size, 0)
target = (50 // grid_size, 750 // grid_size, 0)

path, visited = astar_3d(config_space, source, target)


print('Path length: {}'.format(len(path)))

path_length = 0
for i in range(len(path)-1):
    if path[i][0] == path[i+1][0] and path[i][1] == path[i+1][1]:
        continue
    path_length += 1
print('Path length: {}'.format(path_length))

def get_angle(z):
    angles = np.arange(0, 180, 5)
    return angles[z]

def plot_robot_on_path(work_space, path, robot_size):
    fig, ax = plt.subplots(figsize=(10, 10))
    cmap = plt.cm.gray
    inverse_gray_cmap = LinearSegmentedColormap.from_list('inverse_gray', cmap(
       np.linspace(1, 0, cmap.N)))
    ax.imshow(work_space, cmap=inverse_gray_cmap)

    for idx, point in enumerate(path):
```

```
        if idx % 10 == 0 or idx == len(path) - 1:
            x, y, z = point
            angle = get_angle(z)
            rect = Rectangle((y - robot_size[1] // 2, x - robot_size[0] // 2),
                robot_size[1], robot_size[0], angle=0, color='r')
            rect.set_transform(Affine2D().rotate_deg_around(y, x, angle) + ax.
                transData)
            rect.set_edgecolor('black')
            ax.add_patch(rect)


    plt.title('Robot Path')
    plt.show()

path2d = np.array([(x, y) for x, y, _ in path])
visited2d = np.array([(x, y) for x, y, _ in visited])
visited_mask = np.zeros_like(work_space, dtype=bool)
visited_mask[visited2d[:, 0], visited2d[:, 1]] = True

work_space_with_visited = np.copy(work_space)

work_space_with_visited[visited_mask] = 1


fig, ax = plt.subplots(figsize=(10, 10))
cmap = plt.cm.gray
inverse_gray_cmap = LinearSegmentedColormap.from_list('inverse_gray', cmap(np.
    linspace(1, 0, cmap.N)))
ax.imshow(work_space_with_visited , cmap=inverse_gray_cmap)
ax.plot(path2d[:, 1], path2d[:, 0], 'r')
plt.title('Robot Path, A* search')
plt.show()

plot_robot_on_path(work_space.copy(), path, robot_size=(robot_height,
    robot_width))
```

C Question 3

```
test_cspace = config_space


binary_image = test_cspace.copy()
binary_image[binary_image == 0] = 2
binary_image[binary_image == 1] = 0
binary_image[binary_image == 2] = 1


skeleton = skeletonize_3d(binary_image)
skeleton = skeleton.astype(np.uint32)

skeleton[skeleton == 1] = 2
skeleton[skeleton == 0] = 1
skeleton[skeleton == 2] = 0
```

```
def get_closest_point(skeleton, source):
    skeleton_points = np.array(np.where(skeleton == 0)).T
    skeleton_points = skeleton_points.astype(np.float32)
    source = np.array(source)
    source = source.astype(np.float32)
    distances = np.linalg.norm(skeleton_points - source, axis=1)
    closest_point = skeleton_points[np.argmin(distances)]
    closest_point = closest_point.astype(np.uint32)
    return closest_point

source = (50 // grid_size, 50 // grid_size, 0)
target = (50 // grid_size, 750 // grid_size, 0)

closest_point_source = get_closest_point(skeleton, source)
closest_point_target = get_closest_point(skeleton, target)

closest_point_source = tuple(closest_point_source)
closest_point_target = tuple(closest_point_target)

empty_field = np.zeros_like(skeleton)

path_source_skeleton = astar_3d(empty_field, source, closest_point_source)
path_target_skeleton = astar_3d(empty_field, target, closest_point_target)

for point in path_source_skeleton:
    skeleton[point] = 0

for point in path_target_skeleton:
    skeleton[point] = 0

skeleton_test = skeleton.copy()
skeleton_test[skeleton_test == 0] = 2
skeleton_test[skeleton_test == 1] = 0
skeleton_test[skeleton_test == 2] = 1

fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(111, projection='3d')
ax.voxels(skeleton_test, edgecolor='k')
plt.title('Safe path')
plt.show()

print('Path length: {}'.format(len(path_safe)))
path_length = 0
for i in range(len(path_safe)-1):
    if path_safe[i][0] == path_safe[i+1][0] and path_safe[i][1] == path_safe[i
        +1][1]:
         continue
    path_length += 1
print('Path length: {}'.format(path_length))

path2d = np.array([(x, y) for x, y, _ in path_safe])
fig, ax = plt.subplots(figsize=(10, 10))
```

```
cmap = plt.cm.gray
inverse_gray_cmap = LinearSegmentedColormap.from_list('inverse_gray', cmap(np.
    linspace(1, 0, cmap.N)))
ax.imshow(work_space, cmap=inverse_gray_cmap)
ax.plot(path2d[:, 1], path2d[:, 0], 'r')
plt.title('Robot Path, Safe')
plt.show()

plot_robot_on_path(work_space.copy(), path_safe, robot_size=(robot_height,
    robot_width))
```

D Question 4

```
def connect_lines_in_path(field, path):
    connected_path = []

    def draw_line(x0, y0, x1, y1):
        line_points = []
        dx = abs(x1 - x0)
        dy = abs(y1 - y0)
        sx = 1 if x0 < x1 else -1
        sy = 1 if y0 < y1 else -1
        err = dx - dy

        while x0 != x1 or y0 != y1:
            line_points.append((x0, y0))
            e2 = 2 * err
            if e2 > -dy:
                err -= dy
                x0 += sx
            if e2 < dx:
                err += dx
                y0 += sy

        return line_points

    for i in range(len(path) - 1):
        x0, y0 = path[i]
        x1, y1 = path[i + 1]
        connected_path.extend(draw_line(x0, y0, x1, y1))

    x_last, y_last = path[-1]
    connected_path.append((x_last, y_last))

    return connected_path


def prm_2d(n_nodes, field, source, target):

    sampled_points = []
    sampled_points.append(source)
    sampled_points.append(target)
    while len(sampled_points) < n_nodes:
```

```
        x = np.random.randint(0, field.shape[0])
        y = np.random.randint(0, field.shape[1])
        if field[x, y] == 0:
            sampled_points.append((x, y))


    def is_collision_free_line(field, point1, point2):
        line = np.linspace(point1, point2, 100)
        line = line.astype(np.int32)
        for point in line:
            x, y = point
            if field[x, y] == 1:
                return False
        return True

    G = nx.Graph()
    for point in sampled_points:
        G.add_node(point)

    for point in sampled_points:
        for other_point in sampled_points:
            if point != other_point:
                if not G.has_edge(point, other_point):
                    if is_collision_free_line(field, point, other_point):
                        G.add_edge(point, other_point, weight=np.linalg.norm(np.
                            array(point) - np.array(other_point)))

    path = nx.dijkstra_path(G, source, target)
    return G, path


def plot_prm_graph(G, path, field, n_nodes):
    node_coordinates = {}
    for node in G.nodes():
        x, y = node
        new_coordinates = (y, x)
        node_coordinates[node] = new_coordinates
    nx.draw(G, node_coordinates, with_labels=False, node_size=10, node_color='
        skyblue')

    nx.draw_networkx_edges(G, node_coordinates, edge_color='b', alpha=0.1, width
        =1)

    path_edges = list(zip(path, path[1:]))
    nx.draw_networkx_edges(G, node_coordinates, edgelist=path_edges, edge_color=
        'r', width=2)

    cmap = plt.cm.gray
    inverse_gray_cmap = LinearSegmentedColormap.from_list('inverse_gray', cmap(
        np.linspace(1, 0, cmap.N)))
    plt.imshow(field, cmap=inverse_gray_cmap, origin='upper')
    plt.title("PRM Graph with Best Path for {} Nodes".format(n_nodes))
    plt.show()
```

```
source2d = (50 // grid_size, 50 // grid_size)
target2d = (50 // grid_size, 750 // grid_size)


prm_graph, path_prm = prm_2d(50, config_space[:, :, 0], source2d, target2d)
connected_path = connect_lines_in_path(config_space, path_prm)
print('Path Length 50: {}'.format(len(connected_path)))
plot_prm_graph(prm_graph, path_prm, work_space, 50)

prm_graph, path_prm = prm_2d(100, config_space[:, :, 0], source2d, target2d)
connected_path = connect_lines_in_path(config_space, path_prm)
print('Path Length 100: {}'.format(len(connected_path)))
plot_prm_graph(prm_graph, path_prm, work_space, 100)

prm_graph, path_prm = prm_2d(500, config_space[:, :, 0], source2d, target2d)
connected_path = connect_lines_in_path(config_space, path_prm)
print('Path Length 500: {}'.format(len(connected_path)))
plot_prm_graph(prm_graph, path_prm, work_space, 500)
```

E Question 5

```
def rrm_2d(field, source, target, deltaq=10, max_iter=2000, max_nodes=1000):
    def is_collision_free_line(field, point1, point2):
        line = np.linspace(point1, point2, 100)
        line = line.astype(np.int32)
        for point in line:
            x, y = point
            if field[x, y] == 1:
                return False
        return True

    def get_random_point(field):
        x = np.random.randint(0, field.shape[0])
        y = np.random.randint(0, field.shape[1])
        if field[x, y] == 0:
            return (x, y)
        else:
            return get_random_point(field)


    G = nx.Graph()
    G.add_node(source)


    for i in range(max_iter):
        if len(G.nodes()) >= max_nodes:
            break

        q_rand = get_random_point(field)
        if q_rand in G.nodes():
            continue

        q_near = None
```

```
        min_dist = np.inf

        for node in G.nodes():
            dist = np.linalg.norm(np.array(node) - np.array(q_rand))
            if dist < min_dist:
                min_dist = dist
                q_near = node

        direction = (np.array(q_rand) - np.array(q_near)) / np.linalg.norm(np.
            array(q_rand) - np.array(q_near))

        q_new = q_near + deltaq * direction
        q_new = q_new.astype(np.int32)

        if np.linalg.norm(np.array(q_new)-np.array(target)) < deltaq:
            G.add_node(tuple(q_new))
            G.add_node(target)
            G.add_edge(q_near, tuple(q_new), weight=np.linalg.norm(np.array(
                q_near) - np.array(q_new)))
            G.add_edge(tuple(q_new), target, weight=np.linalg.norm(np.array(
                q_new) - np.array(target)))
            break

        if is_collision_free_line(field, q_near, q_new):
            G.add_node(tuple(q_new))
            G.add_edge(q_near, tuple(q_new), weight=np.linalg.norm(np.array(
                q_near) - np.array(q_new)))

    try:
        path = nx.astar_path(G, source, target)
    except:
        path = []

    return G, path


def plot_rrt_graph(G, path, field):
    node_coordinates = {}
    for node in G.nodes():
        x, y = node
        new_coordinates = (y, x)
        node_coordinates[node] = new_coordinates
    nx.draw(G, node_coordinates, with_labels=False, node_size=10, node_color='
        skyblue')
    number_of_nodes = len(G.nodes())

    nx.draw_networkx_edges(G, node_coordinates, edge_color='b', alpha=0.1, width
        =1)

    path_edges = list(zip(path, path[1:]))
    nx.draw_networkx_edges(G, node_coordinates, edgelist=path_edges, edge_color=
        'r', width=2)

    cmap = plt.cm.gray
```

```
    inverse_gray_cmap = LinearSegmentedColormap.from_list('inverse_gray', cmap(
        np.linspace(1, 0, cmap.N)))
    plt.imshow(field, cmap=inverse_gray_cmap, origin='upper')
    plt.title("RRM Graph with Best Path, {} Nodes".format(number_of_nodes))
    plt.show()


rrm_graph, path_rrt = rrm_2d(config_space[:, :, 0], source2d, target2d, deltaq
    =45, max_iter=2000, max_nodes=100)
# print(path_prm)
# conencted_path_rrt = connect_lines_in_path(config_space, path_rrt)
# print('Path Length RRM: {}'.format(len(conencted_path_rrt)))
plot_rrt_graph(rrm_graph, path_rrt, work_space)

conencted_path_rrt = connect_lines_in_path(config_space, path_rrt)
print('Path Length RRM: {}'.format(len(conencted_path_rrt)))
```