A Robot Navigation Framework

I. Introduction

In this work, leveraging a reconstructed 3D semantic map, the objective is to empower the robot with the capability to navigate towards a predetermined destination or guide it to a specific item through the utilization of the Rapidly-exploring Random Trees (RRT) algorithm.

The initial step involves the conversion of the intricate 3D semantic map into a more manageable 2D scatter map. Subsequently, a path to the desired destination is meticulously planned within the confines of this 2D representation. This path, once established, is then projected back into the 3D simulation environment coordinates, seamlessly integrating the intricacies of the original three-dimensional space.

The final phase of the approach involves the precise control of the robotic agent along the generated path, steering it towards the predetermined destination with a high degree of accuracy and efficiency. This comprehensive methodology seamlessly blends the intricacies of 3D mapping, path planning, and robotic control to achieve the overarching goal of precise and intelligent navigation in a dynamic environment.

II. 2D semantic map construction

With a previously crafted 3D semantic map, leverage the point cloud data's x and z coordinates to ingeniously craft a corresponding 2D map. The process involves constructing a scatter plot, wherein the x-axis represents the z-coordinate of the point cloud data, and the y-axis mirrors the x-coordinate. This strategic approach yields a comprehensive 2D map, encapsulating essential spatial information and facilitating a more accessible representation for subsequent analyses and navigation strategies.

The code is shown as follow:

```
import numpy as np
import open3d as o3d
import argparse
import os
import copy
import time
import math
from sklearn.neighbors import NearestNeighbors
import cv2
import matplotlib.pyplot as plt
```

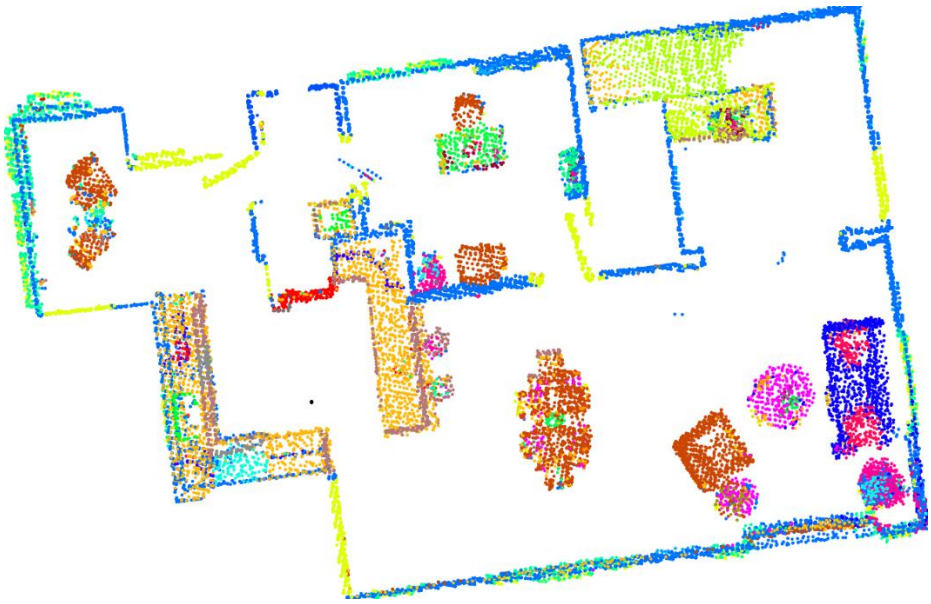
```
class Map:
    def __init__(self, point_path, color_path):
        self.point_path = point_path
        self.color_path = color_path
        self.points = np.load(self.point_path)
        self.colors = np.load(self.color_path)

    def get_pcd(self, pcd_path):
        pcd = o3d.io.read_point_cloud(pcd_path)
        o3d.visualization.draw_geometries([pcd])
        return pcd

    def construct_pcd(self):
        pcd = o3d.geometry.PointCloud()
        self.points = self.points * 10000 / 255
        pcd.points = o3d.utility.Vector3dVector(self.points)
        pcd.colors = o3d.utility.Vector3dVector(self.colors)
        o3d.visualization.draw_geometries([pcd])

        # filter the ceiling and the floor
        xyz_points = np.asarray(pcd.points)
        colors = np.asarray(pcd.colors)
        ceiling_y = 0.0135
        floor_y = -1.35
        filtered_xyz_points = xyz_points[(xyz_points[:, 1] <= ceiling_y)
                                         &(  floor_y <= xyz_points[:, 1])]
        filtered_colors = colors[(xyz_points[:, 1] <= ceiling_y)
                                 &(  floor_y <= xyz_points[:, 1])]
        pcd.points = o3d.utility.Vector3dVector(filtered_xyz_points)
        pcd.colors = o3d.utility.Vector3dVector(filtered_colors)
        o3d.visualization.draw_geometries([pcd])
        return pcd

    def construct_image(self, pcd):
        # create scatter plot
        pixel_per_inches = 1/plt.rcParams['figure.dpi']
        plt.figure(figsize=(1700 * pixel_per_inches,
                            1100 * pixel_per_inches))
```

```
        points = np.asarray(pcd.points)
        plt.scatter(points[:, 2], points[:, 0],s=5, c =
np.asarray(pcd.colors), marker='o')

        # Set the scale
        plt.xlim(points[:, 2].min(), points[:, 2].max())
        plt.ylim(points[:, 0].min(), points[:, 0].max())
        plt.axis('off')
        plt.savefig('map.png', bbox_inches = 'tight', pad_inches = 0)

        plt.show()

        # reverse transform
        # x_restored = u * (points[:, 2].max() - points[:, 2].min()) +
points[:, 2].min()
        # z_restored = (points[:, 0].max() - v) * (points[:, 0].max() -
points[:, 0].min()) / (points[:, 0].max() - points[:, 0].min()) +
points[:, 0].min()


if __name__ == '__main__':
    point_path = "semantic_3d_pointcloud/point.npy"
    color_path = "semantic_3d_pointcloud/color01.npy"
    semantic_map = Map(point_path,color_path)
    pcd = semantic_map.construct_pcd()
    map_points = np.asarray(pcd.points)
    map_colors = np.asarray(pcd.colors)
    np.save("2D_semantic_map_points.npy",map_points)
    np.save("2D_semantic_map_colors.npy",map_colors)
    semantic_map.construct_image(pcd)
```

In this code, the initial step involves specifying the paths for both point data and color data. Subsequently, the construct_pcd() function is employed to create the original point cloud, wherein adjustments such as multiplication by 10000 and division by 255 are applied to seamlessly transition the data into the simulation environment coordinates. To refine the point cloud, filtering processes exclude elements corresponding to the ceiling and floor.

Following this, the construct_image() function orchestrates the generation of a

2D scatter plot. Essential to this process is a nuanced understanding of the display's DPI configuration. The scatter plot dimensions are meticulously set at 1700 x 1100, with the x-axis of the plot representing the z data in the point cloud. Conversely, the y-axis corresponds to the x data in the point cloud. The image boundaries are calibrated to the minimum and maximum values of the data. The resultant 2D map provides a visual representation of the spatial data, offering insights into the distribution and structure of the original point cloud.



Fig(1). 2D semantic map of the scene

III. RRT Algorithm

Code:

With the previous 2D semantic map, we can plan the path on the map directly with the RRT Algorithm. The code is shown in follow:

```
import numpy as np
import argparse
import os
import random
import math
import cv2

class Nodes:
    def __init__(self, x, y):
        self.x = x
```

```
        self.y = y
        self.x_parent = []
        self.y_parent = []

class RRTtree:
    def __init__(self, map, offset, iteration):
        # create black obstacle map
        img = cv2.imread(map, cv2.IMREAD_GRAYSCALE)
        img[np.where((img[:,:] != 255))] = 0
        dilation = cv2.erode(img, np.ones((3,3), np.uint8),
iterations=12)
        cv2.imshow('dilation',dilation)
        cv2.waitKey(0)
        cv2.destroyAllWindows()
        self.dilation = dilation
        self.map = cv2.imread(map)
        self.iteration = iteration
        self.start = [0, 0]
        self.goal = [0, 0]
        self.offset = offset
        self.start_nodes = [()]
        self.target = " "

    def set_target(self, target):
        object_positions = {"refrigerator": (455, 457),
                            "lamp": (894, 657),
                            "cushion": (1039, 454),
                            "rack": (748, 284),
                            "cooktop": (380, 545)}
        x, y = object_positions[target]
        self.target = target
        self.goal = (x, y)

    def random_point(self, height, width):
        random_x = random.randint(0, width)
        random_y = random.randint(0, height)
        return (random_x, random_y)
```

```
    def distance(self, x1, y1, x2, y2):
        return math.sqrt(((x1 - x2) ** 2) + ((y1 - y2) ** 2))

    def angle(self, x1, y1, x2, y2):
        return math.atan2(y2 - y1, x2 - x1)

    def nearest_node(self, x, y, node_list):
        distances = [self.distance(x, y, node.x, node.y) for node in
node_list]
        return distances.index(min(distances))

    def collision(self, x1, y1, x2, y2, img):
        color = []
        if int(x1) == int(x2) and int(y1) == int(y2):
            print("No collision")
            return False

        line_points = np.column_stack((np.linspace(x1, x2, num=100),
np.linspace(y1, y2, num=100)))

        for point in line_points:
            x, y = map(int, point)
            color.append(img[y, x])

        if 0 in color:
            return True
        else:
            return False

    def expansion(self, random_x, random_y, nearest_x, nearest_y,
offset, map):
        theta = self.angle(nearest_x, nearest_y, random_x, random_y)
        new_node_x = nearest_x + offset * np.cos(theta)
        new_node_y = nearest_y + offset * np.sin(theta)
        width, height = map.shape

        if new_node_y < 0 or new_node_y > width or new_node_x < 0 or
new_node_x > height:
```

```
            expand_connect = False
        else:
            if self.collision(new_node_x, new_node_y, nearest_x,
nearest_y, map):
                expand_connect = False
            else:
                expand_connect = True
        return (new_node_x, new_node_y, expand_connect)

    def extend(self, Tree, goal_node, map):
        x = Tree[-1].x
        y = Tree[-1].y
        nearest_index_b = self.nearest_node(x, y, goal_node)
        nearest_x = goal_node[nearest_index_b].x
        nearest_y = goal_node[nearest_index_b].y

        if self.collision(x, y, nearest_x, nearest_y, map):
            extend_connect = False
        else:
            extend_connect = True

        return (extend_connect, nearest_index_b)

    def rrt_path(self, target):
        if target != " ":
            self.set_target(target)
            print("Successfully find the target.")
        else:
            print("No object target.")

        height, width = self.dilation.shape

        self.start_nodes[0] = Nodes(self.start[0], self.start[1])
        self.start_nodes[0].x_parent.append(self.start[0])
        self.start_nodes[0].y_parent.append(self.start[1])

        i = 1
        while i < self.iteration:
```

```
            Tree_A = self.start_nodes.copy()
            random_x, random_y = self.random_point(height, width)

            nearest_index = self.nearest_node(random_x, random_y,
Tree_A)
            nearest_x, nearest_y = Tree_A[nearest_index].x,
Tree_A[nearest_index].y
            new_node_x, new_node_y, expand_connect =
self.expansion(random_x, random_y,
                                                                    nearest
_x, nearest_y,
                                                                    self.of
fset, self.dilation)

            if expand_connect:
                Tree_A.append(Nodes(new_node_x, new_node_y))
                Tree_A[i].x_parent =
Tree_A[nearest_index].x_parent.copy()
                Tree_A[i].y_parent =
Tree_A[nearest_index].y_parent.copy()
                Tree_A[i].x_parent.append(new_node_x)
                Tree_A[i].y_parent.append(new_node_y)

                cv2.circle(self.map, (int(new_node_x), int(new_node_y)),
2,
                           (0, 0, 255), thickness=3, lineType=8)
                cv2.line(self.map, (int(new_node_x), int(new_node_y)),
                         (int(Tree_A[nearest_index].x),
int(Tree_A[nearest_index].y)),
                         (0, 255, 0), thickness=1, lineType=8)
                cv2.imwrite("RRT_Path/" + str(i) + ".jpg", self.map)
                cv2.imshow("image", self.map)
                cv2.waitKey(1)

                extend_connect, index = self.extend(Tree_A,
[Nodes(self.goal[0], self.goal[1])], self.dilation)

                if extend_connect:
```

```
                    print("Path is successfully formulated")
                    path = []
                    cv2.line(self.map, (int(new_node_x),
int(new_node_y)),
                             (int(self.goal[0]), int(self.goal[1])), (0,
255, 0), thickness=1, lineType=8)

                    for i in range(len(Tree_A[-1].x_parent)):
                        path.append((Tree_A[-1].x_parent[i], Tree_A[-
1].y_parent[i]))

                    Nodes(self.goal[0], self.goal[1]).x_parent.reverse()
                    Nodes(self.goal[0], self.goal[1]).y_parent.reverse()

                    for i in range(len(Nodes(self.goal[0],
self.goal[1]).x_parent)):
                        path.append((Nodes(self.goal[0],
self.goal[1]).x_parent[i],
                                     Nodes(self.goal[0],
self.goal[1]).y_parent[i]))

                    # Draw the last segment with a different color (red)
                    cv2.line(self.map, (int(Tree_A[-1].x_parent[-1]),
int(Tree_A[-1].y_parent[-1])),
                             (int(self.goal[0]), int(self.goal[1])),
(255, 0, 0), thickness=2, lineType=8)

                    for i in range(len(path) - 1):
                        cv2.line(self.map, (int(path[i][0]),
int(path[i][1])),
                                 (int(path[i + 1][0]), int(path[i +
1][1])), (255, 0, 0), thickness=2, lineType=8)

                    cv2.waitKey(1)
                    cv2.imwrite("RRT_Path/" + str(i) + ".jpg", self.map)
                    if self.target == " ":
                        cv2.imwrite("rrt.jpg", self.map)
                    else:
```

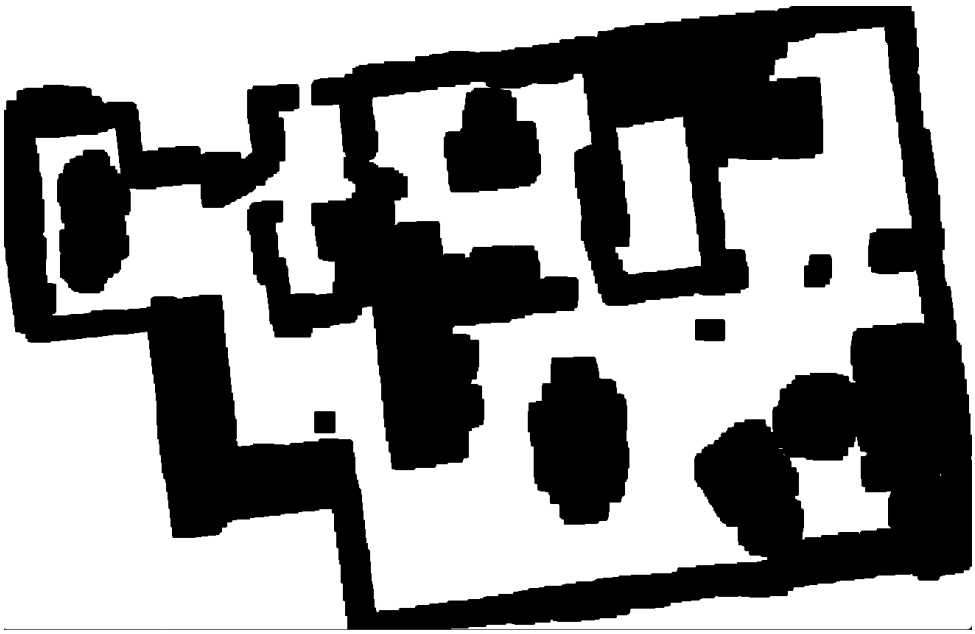
```
                    cv2.imwrite("RRT_Path/" + self.target + ".jpg",
self.map)
                    break
            else:
                continue

            i = i + 1
            self.start_nodes = Tree_A.copy()

        if i == self.iteration:
            print("Failed to find the path")
        print("Number of iteration: ", i)

        path = np.asarray(path)
        width_transform = width / 17
        height_transform = height / 11
        path[:, 0] = path[:, 0] / width_transform - 6
        path[:, 1] = 7 - path[:, 1] / height_transform
        return path

# click on the picture to get the point
def draw_circle(event, u, v, flags, param):
    global coordinates
    if event == cv2.EVENT_LBUTTONDBLCLK:
        cv2.circle(rrt.map, (u, v), 5, (255, 0, 0), -1)
        coordinates.append(u)
        coordinates.append(v)

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Parameters:')
    parser.add_argument('-p', type=str, default='map.png',
metavar='ImagePath',
                        action='store', dest='imagePath',
                        help='File path of the map image')
    parser.add_argument('-o', type=int, default=40, metavar='offset',
                        action='store', dest='offset',
                        help='Step size in RRT algorithm')
    parser.add_argument('-f', type=str, default=" ", metavar='END',
```

```
                        action='store', dest='End',
                        help='The target object of the Navigation')
    args = parser.parse_args()

    try:
        os.system("rm -rf RRT_Path")
    except:
        print("Directory is not exist")
    os.mkdir("RRT_Path")

    target = args.End
    rrt = RRTtree(args.imagePath, args.offset, 10000)

    coordinates = []
    print("Set the start point and goal point by double click on the
image")
    print("Or press ESC to exit")
    cv2.namedWindow('image')
    cv2.setMouseCallback('image', draw_circle)
    while 1:
        cv2.imshow('image', rrt.map)
        k = cv2.waitKey(20) & 0xFF
        if k == 27:
            break
    rrt.start = (coordinates[0], coordinates[1])
    rrt.goal = (coordinates[2], coordinates[3])
    path = rrt.rrt_path(target)
    np.save('path', path)
```

In the "Nodes" class, the coordinates (x, y) of a node are stored along with the parent node's coordinates. This parent node essentially represents the path from the current node to the starting point. By retaining this information for each node, the algorithm can efficiently reconstruct the entire path once the destination is reached. This mechanism facilitates the extraction of a comprehensive path by linking together the nodes that collectively constitute the trajectory from the starting point to the final destination.

In the “RRTtree” class, defines the function and the structure of the path planning procedure:

(1) __init__() : In the constructer function, initialize the class and set the initial values. First, load the map image in gray scale, then binarize the image by setting the intensity to 0 for the pixel that is not while (255), process the image by dilation to enlarge the obstacle for a better quality map, the map for planning the path is shown as follow:



Fig(2). Dilation map to plan the path

Then the class set the step size (offset), initialize the start point and goal point and the target object.

(2) set_target(): In this function, construct a dictionary to set the pixel coordinate of the target object pool, the pixel value is confirm by writing another python code to check the pixel value by clicking on the image, which is shown as follow:

```
import cv2

def click_event(event, x, y, flags, param):
    if event == cv2.EVENT_LBUTTONDOWN:
        print(f"Pixel coordinates (x, y): ({x}, {y})")

if __name__ == "__main__":
    image_path = "map.png"

    # read the image
```

```
    img = cv2.imread(image_path)

    cv2.namedWindow("image")
    cv2.setMouseCallback("image", click_event)

    while True:
        # Show the image
        cv2.imshow("image", img)

        # Press Esc to quit
        if cv2.waitKey(20) & 0xFF == 27:
            break

    cv2.destroyAllWindows()
```

When a target is selected, set the x value and y value of the goal points by the configuration of the object.

(3) random_point(): This function generates a random point on the current map.

(4) distance(): This function calculates the L2 distance between new points and the nearest node.

(5) angle(): This function calculate angle between new point and nearest node

(6) nearest_node(): This function returns the index of the point in the node_list that is closest to the given point (x, y). It effectively identifies and retrieves the index of the nearest node among the set of points stored in node_list, assisting in tasks such as identifying the closest node during the RRT (Rapidly-exploring Random Tree) algorithm's expansion and path planning.

(7) collision(): This function, named **collision**, is responsible for checking whether a collision occurs between two specified points, (x1, y1) and (x2, y2), in the given image represented by **img**. The function discretizes the line segment between the two points into 100 equally spaced points and checks the color of each pixel along this line in the image. If any pixel color is black (indicating an obstacle or collision), the function returns **True**, signifying that a collision has occurred. Otherwise, if no black pixels are encountered, the function returns **False**, indicating that there is no collision along the specified path.

Additionally, the function handles a special case where the two points are the same, in which case it concludes that there is no collision.

This collision-checking mechanism is vital in algorithms like RRT (Rapidly-exploring Random Tree) to ensure that the generated paths do not intersect with

obstacles in the environment.

(8) expansion(): The expansion function in the code is responsible for determining whether a new node, generated randomly around the nearest node, can be added to the tree without causing a collision with obstacles in the environment.

The function checks whether the new node is within the bounds of the environment defined by the obstacle map. If the new node is out of bounds, expand_connect is set to False, indicating that the expansion is not successful.

If the new node is within bounds, the function checks for collisions with obstacles using the collision function. If a collision occurs, expand_connect is set to False; otherwise, it is set to True.

The function returns a tuple (new_node_x, new_node_y, expand_connect), providing the coordinates of the new node and a boolean flag indicating whether the expansion was successful (no collision and within bounds).

(9) extend(): The extend method in the RRTtree class is responsible for checking whether there is a collision between the last node in a given tree (Tree) and a goal node. It returns a tuple containing a boolean value (extend_connect) indicating whether the extension is possible without collision and the index (nearest_index_b) of the nearest node in the goal node list.

(10) rrt_path(): The rrt_path method in the RRTtree class serves as the main function responsible for generating a path using the Rapidly-exploring Random Tree (RRT) algorithm. Let's break down and explain this method in a cohesive paragraph:

Firstly, it checks if a target object is specified. If a target is provided, it sets the goal and prints a success message; otherwise, it prints a message indicating the absence of an object target.

Next, it retrieves the height and width of the image. Subsequently, it initializes the starting node. The main loop runs until the maximum specified number of iterations (self.iteration) is reached.

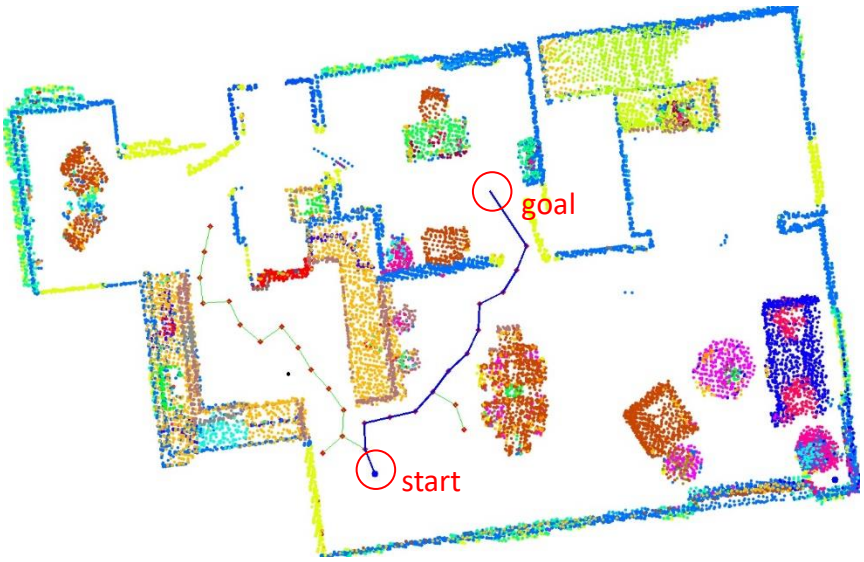
In each iteration:

- It generates a random point.
- Finds the node in the current tree (Tree_A) that is closest to the random point.
- Creates a new node by extending from the nearest node to the random point, performing collision checking.
- If there is no collision, it adds the new node to the tree and updates the map for visualization.
- Checks if the extension reaches the goal node. If successful, it prints a success message and forms the path.

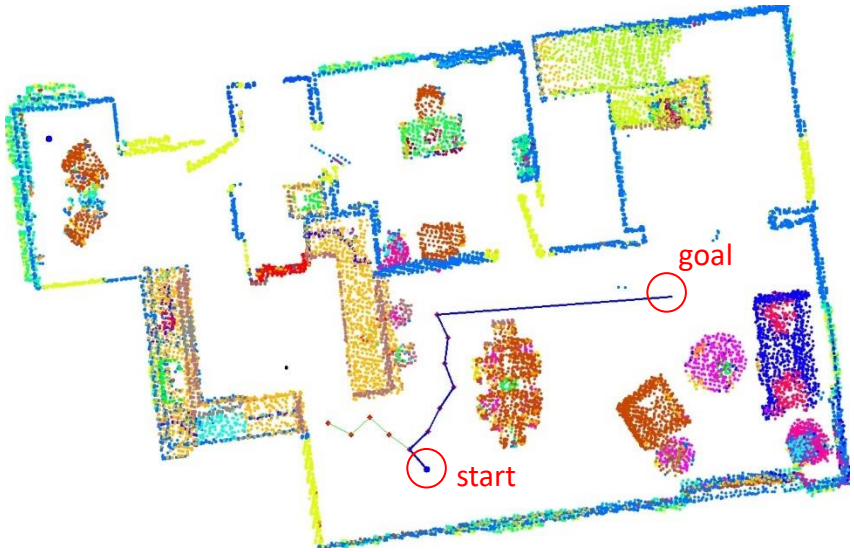
  Finally, the path is transformed to fit the map's dimensions. If the maximum

iteration count is reached without finding a path, it prints a message indicating the failure. The function concludes by returning the generated path. Overall, this function efficiently applies the RRT algorithm, iteratively exploring the search space by randomly expanding the tree to find a viable path.

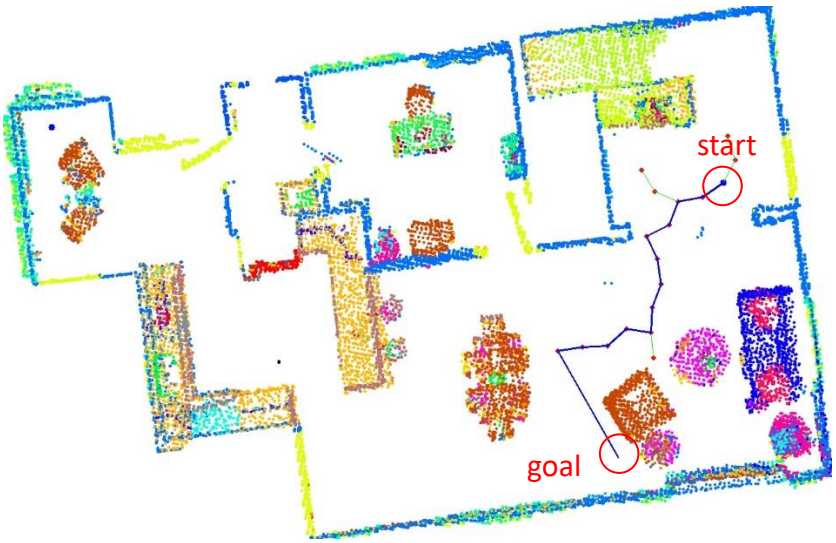
Results:



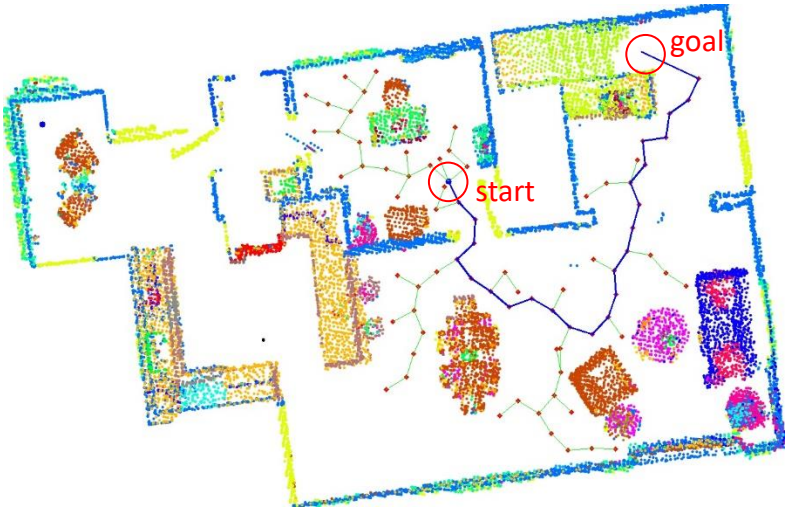
Fig(3). Generated RRT path to the object “rack”



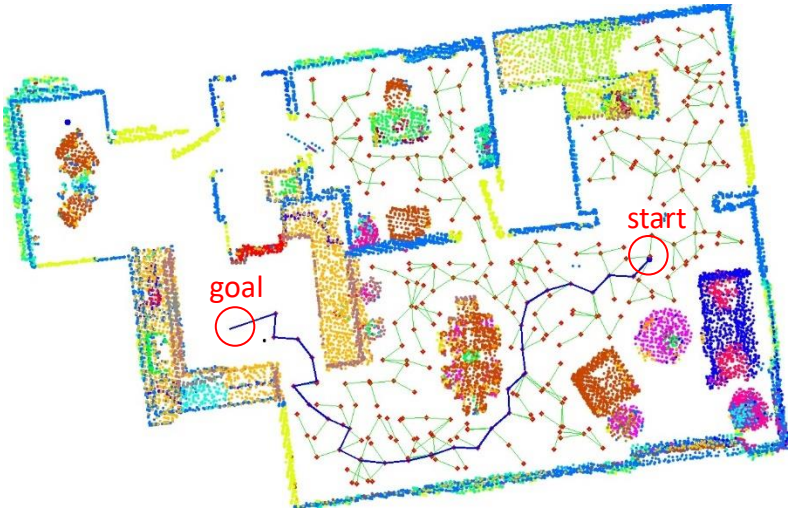
Fig(4). Generated RRT path to the object “cushion”



Fig(5). Generated RRT path to the object “lamp”



Fig(6). Generated RRT path to the object “stair”



Fig(7). Generated RRT path to the object “cooktop”

The images above are the result of the RRT path planning algorithm, the starting point and the goal point are marked on the map. The red dot are the randomly generated nodes, and the green lines are the connection between different nodes. The blue lines are the final path from the starting point to the goal point.

IV. Robot navigation

After planning the path, project the 2D path into 3D simulation environment path, and navigate the agent to the target in Habitat simulation environment. The simulation code is shown as follow:

Code:

```
import numpy as np
from PIL import Image
import numpy as np
import habitat_sim
from habitat_sim.utils.common import d3_40_colors_rgb
import cv2
import json
import math
from mit_semseg.utils import colorEncode
from scipy.io import loadmat
import argparse
import os

# This is the scene we are going to load.
# support a variety of mesh formats, such as .glb, .gltf, .obj, .ply
test_scene = "replica_v1/apartment_0/habitat/mesh_semantic.ply"
json_path = "replica_v1/apartment_0/habitat/info_semantic.json"
colors = loadmat('color101.mat')['colors']
colors = np.insert(colors, 0, values=np.array([[0,0,0]]), axis=0)

# instance id to semantic id
with open(json_path, "r") as f:
    annotations = json.load(f)

id_to_label = []
instance_id_to_semantic_label_id = np.array(annotations["id_to_label"])
for i in instance_id_to_semantic_label_id:
```

```
    if i < 0:
        id_to_label.append(0)
    else:
        id_to_label.append(i)
id_to_label = np.asarray(id_to_label)

sim_settings = {
    "scene": test_scene,  # Scene path
    "default_agent": 0,  # Index of the default agent
    "sensor_height": 1.5,  # Height of sensors in meters, relative to
the agent
    "width": 512,  # Spatial resolution of the observations
    "height": 512,
    "sensor_pitch": 0,  # sensor pitch (x rotation in rads)
}

def rgb_to_bgr(image):
    return image[:, :, [2, 1, 0]]

def transform_depth(image):
    depth_img = (image / 10 * 255).astype(np.uint8)
    return depth_img

def transform_semantic(semantic_obs):
    semantic_img = Image.new("P", (semantic_obs.shape[1],
semantic_obs.shape[0]))
    semantic_img.putpalette(colors.flatten())
    semantic_img.putdata(semantic_obs.flatten().astype(np.uint8))
    semantic_img = semantic_img.convert("RGB")
    semantic_img = cv2.cvtColor(np.asarray(semantic_img),
cv2.COLOR_RGB2BGR)
    return semantic_img

# This function generates a config for the simulator.
# It contains two parts:
# one for the simulator backend
# one for the agent, where you can attach a bunch of sensors
def make_simple_cfg(settings):
```

```
    # simulator backend
    sim_cfg = habitat_sim.SimulatorConfiguration()
    sim_cfg.scene_id = settings["scene"]

    # In the 1st example, we attach only one sensor,
    # a RGB visual sensor, to the agent
    rgb_sensor_spec = habitat_sim.CameraSensorSpec()
    rgb_sensor_spec.uuid = "color_sensor"
    rgb_sensor_spec.sensor_type = habitat_sim.SensorType.COLOR
    rgb_sensor_spec.resolution = [settings["height"], settings["width"]]
    rgb_sensor_spec.position = [0.0, settings["sensor_height"], 0.0]
    rgb_sensor_spec.orientation = [
        settings["sensor_pitch"],
        0.0,
        0.0,
    ]
    rgb_sensor_spec.sensor_subtype = habitat_sim.SensorSubType.PINHOLE

    #depth snesor
    depth_sensor_spec = habitat_sim.CameraSensorSpec()
    depth_sensor_spec.uuid = "depth_sensor"
    depth_sensor_spec.sensor_type = habitat_sim.SensorType.DEPTH
    depth_sensor_spec.resolution = [settings["height"],
settings["width"]]
    depth_sensor_spec.position = [0.0, settings["sensor_height"], 0.0]
    depth_sensor_spec.orientation = [
        settings["sensor_pitch"],
        0.0,
        0.0,
    ]
    depth_sensor_spec.sensor_subtype = habitat_sim.SensorSubType.PINHOLE

    #semantic snesor
    semantic_sensor_spec = habitat_sim.CameraSensorSpec()
    semantic_sensor_spec.uuid = "semantic_sensor"
    semantic_sensor_spec.sensor_type = habitat_sim.SensorType.SEMANTIC
    semantic_sensor_spec.resolution = [settings["height"],
settings["width"]]
```

```
    semantic_sensor_spec.position = [0.0, settings["sensor_height"],
0.0]
    semantic_sensor_spec.orientation = [
        settings["sensor_pitch"],
        0.0,
        0.0,
    ]
    semantic_sensor_spec.sensor_subtype =
habitat_sim.SensorSubType.PINHOLE

    # agent
    agent_cfg = habitat_sim.agent.AgentConfiguration()
    agent_cfg.sensor_specifications = [rgb_sensor_spec,
depth_sensor_spec, semantic_sensor_spec]
    agent_cfg.action_space = {
        "move_forward": habitat_sim.agent.ActionSpec(
            "move_forward", habitat_sim.agent.ActuationSpec(amount=0.01)
# 0.01 => 0.01 m
        ),
        "turn_left": habitat_sim.agent.ActionSpec(
            "turn_left", habitat_sim.agent.ActuationSpec(amount=1.0) #
1.0 => 1 degree
        ),
        "turn_right": habitat_sim.agent.ActionSpec(
            "turn_right", habitat_sim.agent.ActuationSpec(amount=1.0)
        ),
    }

    return habitat_sim.Configuration(sim_cfg, [agent_cfg])

cfg = make_simple_cfg(sim_settings)
sim = habitat_sim.Simulator(cfg)

# initialize an agent
agent = sim.initialize_agent(sim_settings["default_agent"])

# Set agent state
path = np.load('path.npy') #load the path
```

```
start = path[0]
print(start)
agent_state = habitat_sim.AgentState()
agent_state.position = np.array([start[1], 0.0, start[0]])  # agent in
world space
agent.set_state(agent_state)

# obtain the default, discrete actions that an agent can perform
# default action space contains 3 actions: move_forward, turn_left, and
turn_right
action_names =
list(cfg.agents[sim_settings["default_agent"]].action_space.keys())
print("Discrete action space: ", action_names)

def navigateAndSee(action=""):
    if action in action_names:
        observations = sim.step(action)
        #print("action: ", action)

        RGB_img = rgb_to_bgr(observations["color_sensor"])
        SEIMEN_img
=  transform_semantic(id_to_label[observations["semantic_sensor"]])
        index =
np.where((SEIMEN_img[:,:,0]==b)*(SEIMEN_img[:,:,1]==g)*(SEIMEN_img[:,:,
2]==r))
        if len(index[0]) != 0:
            RGB_img[index] = cv2.addWeighted(RGB_img[index], 0.6,
SEIMEN_img[index], 0.4, 50)
        cv2.imshow("RGB", RGB_img)
        cv2.waitKey(1)
        videowriter.write(RGB_img)
        agent_state = agent.get_state()
        sensor_state = agent_state.sensor_states['color_sensor']
        print("camera pose: x y z rw rx ry rz")
        print(sensor_state.position[0],sensor_state.position[1],sensor_s
tate.position[2],  sensor_state.rotation.w, sensor_state.rotation.x,
sensor_state.rotation.y, sensor_state.rotation.z)
        return sensor_state
```

```
def driver(pre_node,start,end):
    # rotate
    print("ok")
    if pre_node == []:
        v1 =np.array([-1,0])
    else:
        v1 = np.array([start[0]-pre_node[0],start[1]-pre_node[1]])
    v2 = np.array([end[0]-start[0],end[1]-start[1]])
    print(v1,v2)
    flag = v1[0]*v2[1]-v1[1]*v2[0]
    value = v1@v2
    v1 = math.sqrt((v1[0])**2+(v1[1])**2)
    v2 = math.sqrt((v2[0])**2+(v2[1])**2)

    goal_ry = int(math.acos(value/(v1*v2))/math.pi*180)
    print(goal_ry)
    print("rotate number",int(goal_ry))
    if(flag>=0):
        action = "turn_left"
    else:
        action = "turn_right"
    for i in range(int(abs(goal_ry))):
        sensor_state = navigateAndSee(action)

    #forword
    action = "move_forward"
    forward_distance = math.sqrt((end[0]-start[0])**2+(end[1]-
start[1])**2)
    step = int(forward_distance/0.01)
    for i in range(step):
        sensor_state = navigateAndSee(action)
    x = sensor_state.position[0]
    z = sensor_state.position[2]

    return (z,x)

pre_node = []
```

```
start = path[0]
goal = {"refrigerator":(255, 0, 0),
        "rack":(0, 255, 133),
        "cushion":(255, 9, 92),
        "lamp":(160, 150, 20),
        "stair":(173,255,0),
        "cooktop":(7, 255, 224)}

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description = 'Below are the
params:')
    parser.add_argument('-f', type=str, default=" ",metavar='END',
action='store', dest='End',
                help='Where want to go')
    args = parser.parse_args()

    # save video initial
    if not os.path.exists("video/"):
        os.makedirs("video/")
    video_path = "video/" + args.End + ".mp4"
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    videowriter = cv2.VideoWriter(video_path, fourcc, 100, (512, 512))

    end_rgb = goal[args.End]
    r = end_rgb[0]
    g = end_rgb[1]
    b = end_rgb[2]
    for i in range(len(path)-1):
        temp = start
        start = driver(pre_node,start,path[i+1])
        pre_node = temp
    videowriter.release()
```

The code utilizes the Habitat-Sim library to simulate an agent navigating through a 3D environment, generating video output of the agent's perspective, and incorporating semantic information into the visualizations:

(1) **Libraries and Modules:** First imports necessary libraries, including NumPy,

Pillow (PIL), habitat_sim, OpenCV (cv2), JSON, math, argparse, and os.

(2) **Scene Setup:** Specifies the 3D scene (test_scene) and loads associated semantic information from a JSON file (json_path). Defines color mappings using MATLAB color data loaded from a .mat file. In preparation for utilization within the simulation environment, it is imperative to apply colorization to labeled images using the color information from the color101 dataset. Additionally, it has been observed that an extra color entry is required when importing the color data to ensure accurate color mapping

(3) **Simulator Configuration:** Configures the simulator settings, such as the scene, default agent, sensor specifications (RGB, depth, semantic), and agent actions.

(4) **Configuration Generation:** Defines a function (make_simple_cfg) to generate a configuration for the simulator and agent.

(5) **Simulator Initialization:** Initializes the simulator using the provided configuration settings.

(6) **Agent Initialization:** Creates an agent within the simulator and sets its initial state based on a pre-determined path (path.npy).

(7) **Navigation and Visualization:** Implements functions for converting RGB images to BGR format, transforming depth images, and rendering semantic images.

In function navigationAndSee(), This function is responsible for performing a navigation action in the simulation environment and updating the visual display. It takes an action as input, such as "move_forward", "turn_left", or "turn_right," and then executes this action in the simulator. After the action, it retrieves observations from the simulation, including color and semantic sensor data.

In function driver(), This function drives the agent from a starting position to an end position in the simulation environment. It involves rotating the agent based on the difference between the current and target directions and then moving the agent forward to the target position. First, calculate the direction vectors v1 and v2. Secondly, determine the rotation direction (left or right) based on the cross product of v1 and v2. Calculate the rotation angle (goal_ry) using the dot product and magnitudes of v1 and v2. With the obtained angle, execute rotation actions to align the agent with the target direction, then calculate the forward distance to the target. Execute forward movement actions to reach the target position and the function retrieve and return the final position coordinates (z, x).

(8) **Goal Specification and Video Recording:** Defines a dictionary (goal) mapping object names to RGB colors. Iterates through the provided path, directing the agent to navigate from its current position to the next target position. The agent's movements are recorded, and the resulting video is saved.

(9) **Command-Line Argument Handling:** Parses command-line arguments,

allowing the user to specify the target object for navigation.

(10) **Video Output:** Saves the recorded navigation sequence as an MP4 video file in the "video/" directory.

Discussion:

In the navigation process, designating the point directly in front of the target object as the goal requires the agent to first align its orientation with the next waypoint before proceeding to move forward. It's important to note that the size of the agent is not known during the project, and the path planning stage does not account for the robot's dimensions. Consequently, there is a risk of collision with the environment due to the undisclosed size of the robot.

In the path planning phase, the step size of the Rapidly Exploring Random Trees (RRT) algorithm plays a crucial role. Choosing an appropriate step size is essential because an excessively large step size can compromise the quality of the generated path. On the other hand, an overly small step size results in an excessive number of waypoints, leading to an inefficient path.

Finding the right balance in the step size is imperative to ensure that the generated path is both collision-free and efficient. This consideration is particularly crucial given the unknown size of the robot. Striking the optimal balance in step size allows the agent to navigate the environment effectively while avoiding collisions and maintaining the quality of the path.

V. Bi-RRT

To enhance the efficiency and performance of the original RRT algorithm, a modification involves implementing a Bit RRT (Rapidly Exploring Random Trees) variant. The Bit RRT algorithm introduces a binary space partitioning technique, utilizing a bit representation to efficiently encode the configuration space. This modification aims to reduce the dimensionality of the search space, thereby accelerating the exploration process.

By leveraging the Bit RRT approach, the algorithm can benefit from a more compact representation of the configuration space, leading to faster nearest-neighbor searches and improved exploration efficiency. The bit encoding helps capture the topological features of the environment with a reduced memory footprint. The modify code is shown as follow:

```
import numpy as np
import argparse
import os
import random
import math
import cv2
import matplotlib.pyplot as plt

class Nodes:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.x_parent = []
        self.y_parent = []

class RRTree:
    def __init__(self, map, offset, iteration):
        # create black obstacle map
        img = cv2.imread(map, cv2.IMREAD_GRAYSCALE)
        img[np.where((img[:,:] != 255))] = 0
        dilation = cv2.erode(img, np.ones((3,3), np.uint8), iterations =
12)
        # cv2.imshow('dilation',dilation)
        # cv2.waitKey(0)
        # cv2.destroyAllWindows()
        self.dilation = dilation
        self.map = cv2.imread(map)
        self.iteration = iteration
        self.start = [0, 0]
        self.goal = [0, 0]
        self.offset = offset
        self.start_nodes = [()]
        self.goal_nodes = [()]
        self.target = " "

    def set_target(self,target):
        object = {"refrigerator":(455,457),
                  "rack":(748,284),
```

```
                "cushion":(1039,454),
                "lamp":(977,716),
                "stair":(1075,80),
                "cooktop":(380,545)}
        x = object[target][0]
        y = object[target][1]
        self.target = target
        self.goal = (x,y)

    # generate a random point in the 2D image pixel
    def random_point(self, height, width):
        random_x = random.randint(0, width)
        random_y = random.randint(0, height)
        return (random_x,random_y)

    # Calculate L2 distance between new point and nearest node
    def distance(self,x1,y1,x2,y2):
        return math.sqrt(((x1-x2) ** 2 ) + ((y1-y2) ** 2))

    # Calculate angle between new point and nearest node
    def angle(self,x1,y1,x2,y2):
        return math.atan2(y2-y1, x2-x1)

    # return the index of point among the points in node_list,
    # the one that closet to point (x,y)
    def nearest_node(self, x, y, node_list):
        distances = [self.distance(x, y, node.x, node.y)
                     for node in node_list]
        return distances.index(min(distances))

    # check collision
    def collision(self, x1, y1, x2, y2, img):
        color = []
        print("check collision between x:", x1, x2)

        if int(x1) == int(x2) and int(y1) == int(y2):
            # Points are the same, no collision
            print("No collision")
```

```
            return False

        line_points = np.column_stack((np.linspace(x1, x2, num=100),
np.linspace(y1, y2, num=100)))

        for point in line_points:
            x, y = map(int, point)
            color.append(img[y, x])

        # If there is black in color => collision
        if 0 in color:
            return True
        else:
            return False

    # check the collision with obstacle and the trajectory
    # if there is no collision, add the newpoint to the tree
    def expansion(self, random_x ,random_y ,nearest_x ,nearest_y ,
offset, map):

        theta = self.angle(nearest_x, nearest_y, random_x, random_y)
        new_node_x = nearest_x + offset * np.cos(theta)
        new_node_y = nearest_y + offset * np.sin(theta)
        width, height = map.shape

        # Check the point is out of bounds or not
        if new_node_y < 0 or new_node_y > width or new_node_x < 0 or
new_node_x > height:
            # print("Points out of bounds")
            expand_connect = False
        else:
            if
self.collision(new_node_x,new_node_y,nearest_x,nearest_y,map):
                expand_connect = False
            else:
                expand_connect = True
        return(new_node_x, new_node_y , expand_connect)
```

```
    # find the neareast point to from
    # newest point in Tree_A to Tree_B
    def extend(self, Tree_A, Tree_B, map):
        # index -1 gets the last object in the list
        x = Tree_A[-1].x
        y = Tree_A[-1].y
        nearest_index_b = self.nearest_node(x,y,Tree_B)
        nearest_x = Tree_B[nearest_index_b].x
        nearest_y = Tree_B[nearest_index_b].y
        # check direct connection
        if self.collision(x,y,nearest_x,nearest_y,map):
            extend_connect = False
        else:
            extend_connect =True

        # return(directCon,nearest_index_a,nearest_index_b)
        return(extend_connect,nearest_index_b)

    def rrt_path(self, target):
        # object Target
        if(target != " "):
            self.set_target(target)
            print("Sucessfully find the target.")
        # point Target
        else:
            print("No object target.")

        height, width = self.dilation.shape

        # record the initial nodes
        self.start_nodes[0] = Nodes(self.start[0], self.start[1])
        self.start_nodes[0].x_parent.append(self.start[0])
        self.start_nodes[0].y_parent.append(self.start[1])
        self.goal_nodes[0] = Nodes(self.goal[0],self.goal[1])
        self.goal_nodes[0].x_parent.append(self.goal[0])
        self.goal_nodes[0].y_parent.append(self.goal[1])

        grow_tree = -1
```

```
        # grow_tree = -1 : spread tree from start to goal
        # grow_tree = 1 : spread tree from goal to start
        # Tree from A to B
        i = 1
        while i < self.iteration:
            if grow_tree == -1:
                Tree_A = self.start_nodes.copy()
                Tree_B = self.goal_nodes.copy()
            else:
                Tree_A = self.goal_nodes.copy()
                Tree_B = self.start_nodes.copy()

            # generate random points
            random_x, random_y = self.random_point(height, width)


            nearest_index = self.nearest_node(random_x,random_y,Tree_A)
            nearest_x, nearest_y = Tree_A[nearest_index].x,
Tree_A[nearest_index].y
            new_node_x,new_node_y,expand_connect =
self.expansion(random_x,random_y,
                                           nearest_x,nearest_y,
                                           self.offset,self.dilation)

            if expand_connect:
                # Add the point to the Tree
                Tree_A.append(Nodes(new_node_x,new_node_y))
                Tree_A[i].x_parent =
Tree_A[nearest_index].x_parent.copy()
                Tree_A[i].y_parent =
Tree_A[nearest_index].y_parent.copy()
                Tree_A[i].x_parent.append(new_node_x)
                Tree_A[i].y_parent.append(new_node_y)

                # display
                cv2.circle(self.map, (int(new_node_x),int(new_node_y)),
2,
                           (0,0,255), thickness=3, lineType=8)
```

```
                cv2.line(self.map, (int(new_node_x),int(new_node_y)),
                         (int(Tree_A[nearest_index].x),
                          int(Tree_A[nearest_index].y)),
                          (0,255,0), thickness=1, lineType=8)
                cv2.imwrite("bi-RRT_Path/"+str(i)+".jpg",self.map)
                cv2.imshow("image",self.map)
                cv2.waitKey(1)

                # if extend successful connect
                # which means the two Trees are succesfully connected
                extend_connect, index =
self.extend(Tree_A,Tree_B,self.dilation)

                if extend_connect :
                    print("bi-RRT Path is successfully formulated")
                    path = []
                    cv2.line(self.map,
                             (int(new_node_x),int(new_node_y)),
                             (int(Tree_B[index].x),int(Tree_B[index].y)),
                             (0,255,0), thickness=1, lineType=8)
                    if grow_tree == -1:
                        for i in range(len(Tree_A[-1].x_parent)):
                            path.append((Tree_A[-1].x_parent[i],Tree_A[-
1].y_parent[i]))
                            pass
                        Tree_B[index].x_parent.reverse()
                        Tree_B[index].y_parent.reverse()
                        for i in range(len(Tree_B[index].x_parent)):
                            path.append((Tree_B[index].x_parent[i],Tree_B
[index].y_parent[i]))
                    else:
                        for i in range(len(Tree_B[index].x_parent)):
                            path.append((Tree_B[index].x_parent[i],Tree_B
[index].y_parent[i]))
                            pass
                        Tree_A[-1].x_parent.reverse()
                        Tree_A[-1].y_parent.reverse()
                        for i in range(len(Tree_A[-1].x_parent)):
```

```
                        path.append((Tree_A[-1].x_parent[i],Tree_A[-
1].y_parent[i]))
                    for i in range(len(path)-1):
                        cv2.line(self.map,
(int(path[i][0]),int(path[i][1])),
(int(path[i+1][0]),int(path[i+1][1])), (255,0,0), thickness=2,
lineType=8)
                    cv2.waitKey(1)
                    cv2.imwrite("bi-RRT_Path/"+str(i)+".jpg",self.map)
                    if(self.target == " "):
                        cv2.imwrite("birrt.jpg",self.map)
                    else:
                        cv2.imwrite("bi-
RRT_Path/"+self.target+".jpg",self.map)
                    break
            else:
                continue

            if grow_tree == -1:
                grow_tree = grow_tree*(-1)
                self.start_nodes = Tree_A.copy()
                self.goal_nodes = Tree_B.copy()
            else:
                grow_tree = grow_tree*(-1)
                i = i+1
                self.start_nodes = Tree_B.copy()
                self.goal_nodes = Tree_A.copy()

        if i==self.iteration:
            print("Failed to find the path")
        print("Number of iteration: ",i*2)

        path = np.asarray(path)
        width_transform = width / 17
        height_transform = height / 11
        path[:,0] = path[:,0] / width_transform - 6
        path[:,1] = 7 - path[:,1] / height_transform
        return path
```

```
# click on the picture to get the point
def draw_circle(event,u,v,flags,param):
    global coordinates
    if event == cv2.EVENT_LBUTTONDBLCLK:
        cv2.circle(rrt.map,(u,v),5,(255,0,0),-1)
        coordinates.append(u)
        coordinates.append(v)

if __name__ == '__main__':

    parser = argparse.ArgumentParser(description = 'Parameters:')
    parser.add_argument('-p', type=str,
default='map.png',metavar='ImagePath',
                        action='store', dest='imagePath',
                        help='File path of the map image')
    parser.add_argument('-o', type=int, default=40, metavar='offset',
                        action='store', dest='offset',
                        help='Step size in RRT algrithm')
    parser.add_argument('-f', type=str, default=" ",metavar='END',
                        action='store', dest='End',
                        help='The target object of the Navigation')
    args = parser.parse_args()

    # remove previously stored data
    try:
      os.system("rm -rf bi-RRT_Path")
    except:
      print("Directory is not exist")
    os.mkdir("bi-RRT_Path")

    target = args.End
    rrt = RRTree(args.imagePath, args.offset, 10000)

    coordinates=[]
    print("Set the start point and goal point by double click on the
image")
    print("Or press ESC to exit")
```

```
cv2.namedWindow('image')
cv2.setMouseCallback('image', draw_circle)
while(1):
    cv2.imshow('image',rrt.map)
    k = cv2.waitKey(20) & 0xFF
    if k == 27:
        break
rrt.start=(coordinates[0],coordinates[1])
rrt.goal=(coordinates[2],coordinates[3])
path = rrt.rrt_path(target)
np.save('path', path)
```

Comparison:

Fig(8). Generated RRT path to the object “refrigerator” with total 82 points

Fig(9). Generated bi-RRT path to the object “refrigerator” with total 46 points

The results demonstrate the enhanced efficiency of the Bi-RRT (Bit Rapidly Exploring Random Trees) method in the context of path planning. Notably, the Bi-RRT approach exhibits a reduction in the total number of waypoints, accompanied by a decrease in the frequency of randomly sampled points along the planned trajectory. This optimization contributes to a faster convergence of the path planning process.

The advantages of the Bi-RRT method become apparent as it effectively streamlines the exploration of the configuration space, resulting in a more concise and efficient representation of the robot's trajectory. The diminished reliance on random samples, coupled with a reduced overall waypoint count, signifies a notable improvement in the algorithm's ability to navigate the environment swiftly and converge towards a solution.

These findings underscore the efficacy of the Bi-RRT method as a path planning strategy, showcasing its potential to outperform traditional RRT algorithms in terms of computational efficiency and convergence speed.