Perception and Decision Making in Intelligent Systems
Homework 2

313554024 林慧旻

Implementation (50%)

a. Code

Detailed explanation of your implementation.
For example:

- **Part 1: 2D semantic map construction**

```
def plot_points(point, pcd, pcd_color):
    combined_coordinates = np.vstack((pcd, point))
    combined_colors = np.vstack((pcd_color, np.array([0, 0, 0], dtype=np.float32)))
    plt.figure()
    plt.scatter(combined_coordinates[:, 2], combined_coordinates[:, 0], c=combined_colors, s=0.25)
    plt.axis('off')
    plt.gca().set_aspect('equal', adjustable='box')
    plt.savefig('map.png', dpi=300, bbox_inches='tight')

    image = cv2.imread('map.png')
    black_pixels = np.all(image == [0, 0, 0], axis=-1)
    pixel_indices = np.where(black_pixels)
    pixel_indices = np.transpose(pixel_indices)
    average_position = np.mean(pixel_indices, axis=0)

    return average_position

def get_trans_matrix():
    point_cloud = np.load('./semantic_3d_pointcloud/point.npy')
    colors = np.load('./semantic_3d_pointcloud/color01.npy')

    mask_roof = point_cloud[:, 1] < -0.03
    mask_floor = point_cloud[:, 1] > -0.001
    mask_others = ~(mask_roof | mask_floor)

    point_cloud = point_cloud[mask_others]
    colors = colors[mask_others]

    reference_points = np.float32([[0.15, 0, 0], [0.05, 0, 0], [-0.05, 0, 0.1], [0, 0, -0.1]])
    transformed_points = [plot_points(pt, point_cloud, colors) for pt in reference_points]

    src_points = np.float32([[pt[0], pt[1]] for pt in transformed_points])
    dst_points = np.float32([[pt[0], pt[2]] for pt in reference_points])

    transformation_matrix, _ = cv2.findHomography(src_points, dst_points)


    scaled_transformation_matrix = transformation_matrix * 10000 / 255

    return scaled_transformation_matrix
```

1. **plot_points**:
    a. **Combine Coordinates**: Stacks the input point with the entire point cloud. This is essentially merging the list of all points with the new point to be highlighted.
    b. **Combine Colors**: Similarly, it stacks the color of the input point (black) with the colors of the entire point cloud.
    c. **Plotting**: Uses `matplotlib` to create a scatter plot of the z and x coordinates and the new added black point. It then saves this plot as an image ‘map.png’.
    d. **Identify Black Pixels**: Loads the image using OpenCV and identifies all pixels that are completely black.
    e. **Average Position**: Calculates the average position of these black pixels in the image and return it as the destination point.
2. **get_trans_matrix:**

a. **Load Data**

b. **Filter Points**: Only leave the points not in the range of 'roof' and 'floor'

c. **Define Reference Points**: Add a set of random points for which the transformation matrix needs to be computed.

d. **Project and Plot**: For each reference point, use `plot_points` to project it onto the 2D image, plot it, and find the average position of the resulting point in the image.

e. **Compute Homography**: Uses OpenCV's `findHomography` to find a transformation matrix that maps the source points in the image to the destination points in the 3D projection.

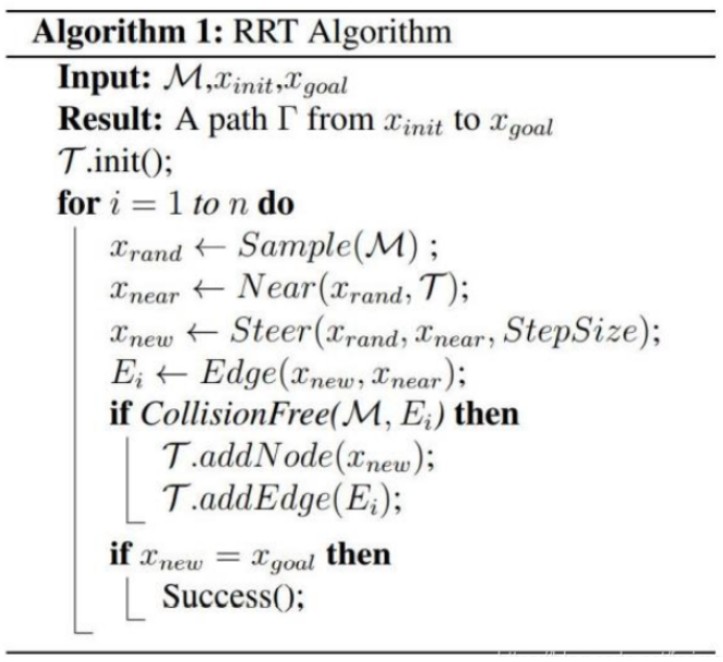
f. **Scale Transformation Matrix**: Adjusts the transformation matrix to scale from pixel coordinates to real-world coordinates.

- **Part 2: RRT Algorithm**

```
    def search(self, target_coor):
        print('---Start RRT Algorithm---')
        print('Start point: ', self.start_p)
        print('Target color: ', self.target)

        consecutive_no_progress = 0
        for i in range(self.max_iter):
            p_rand = self.get_randpoint()
            p_nearNode, p_near = self.find_nearest(p_rand)
            p_new = self.steer(p_near, p_rand)

            if self.collision_check(p_near, p_new):
                consecutive_no_progress += 1
                if consecutive_no_progress > 10:
                    print("No progress made, terminating early.")
                    return None, self.tree
                continue

            consecutive_no_progress = 0
            tmpNode = self.Update_all(p_new, p_nearNode)

            if self.target_check(p_new, target_coor):
                print('Target is found.')
                self.Update_all(target_coor, tmpNode)
                path = []
                tmp = self.tree[-1]
                while True:
                    path.append([tmp.x, tmp.y])
                    tmp = tmp.parent
                    if tmp is None:
                        break
                return path, self.tree

        print('No path was found, repeat RRT again.')
        return None, self.tree
```

1. **Initialization:**
   - `search(target_coor)` starts the RRT search algorithm to find a path from the start point to the target.
   - It initializes `consecutive_no_progress` to track unsuccessful attempts.
2. **RRT Iterations:**

   Pseudo Code of RRT is as belows:



- The algorithm runs for a maximum of `self.max_iter` iterations.
- In each iteration:
  - **Generate random point:** Calls `get_randpoint()` to randomly generate a point in the map.
  - **Find nearest point:** Calls `find_nearest(p_rand)` to find the nearest existing point in the tree using the KD-Tree structure.
  - **Steer towards the random point:** Calls `steer(p_near, p_rand)` to compute a new point (`p_new`) that is a step towards the random point from the nearest point, with a maximum distance defined by `self.step`.

3. **Collision Check:**
   - The new point (`p_new`) is checked for collisions using `collision_check(p_near, p_new)`. It selects a certain numbers of points on the line (p_near, p_new) to see if any of them collides with an obstacle (non-white pixel in the map). Then, the algorithm continues to the next iteration.
   - If no progress is made for more than 10 consecutive steps, the algorithm terminates early.
4. **Update the Tree:**
   - If `p_new` is valid (no collision), it is added to the tree and the KD-Tree is updated by calling `Update_all(p_new, p_nearNode)`.
5. **Target Check:**
   - After updating the tree, it checks if the new point is close to the target using `target_check(p_new, target_coor)`. If the new point is within the `window_size` of the target, the target is considered found.
6. **Path Construction:**
   - When the target is reached, it constructs the path by backtracking from the target node to the start node using the parent references of each node. The path is stored in reverse order, from target to start, and is returned along with the tree.
7. **Completion:**
   - If the maximum iterations are reached without finding a path, the algorithm outputs "No path was found" and returns `None` and the current tree.

Key Points from Other Functions:

- `get_randpoint()`: Generates a random point in the map.
- `find_nearest()`: Uses a KD-Tree to find the closest point in the tree to a random point.
- `steer()`: Moves from the nearest point toward the random point by a fixed step distance.
- `collision_check()`: Verifies if the line between two points collides with any obstacles.
- `Update_all()`: Adds a new point and node to the KD-Tree and tree structure.
- `target_check()`: Verifies if the new point is close enough to the target. Target points are selected manually by `label_target.py`

8. **Target Points**

   I select 2 target points as the last 2 points for each path of each object to make sure the camera can film in front of the object in the end while navigation.

   Ex. cooktop

Only the first target point will be shown on the path. The second one is used for nabigation only.

- **Part 3: Robot navigation**

```
def calculate_direction(sensor_state, target_position):
    diff = np.array([target_position[0] - sensor_state.position[0], target_position[1] - sensor_state
    direction = diff / np.linalg.norm(diff)
    return direction, np.linalg.norm(diff)


def rotate_agent(agent, direction, sim, action_names, frame_idx, id_to_label, target_semantic_id, obj
    deg_diff, action = rotate(agent, direction)
    print(f'{action}: {deg_diff}')

    while deg_diff > 0:
        navigate_and_see(sim, action, action_names, frame_idx, id_to_label, target_semantic_id, obj)
        deg_diff -= turn_deg
        frame_idx += 1

    return frame_idx


def move_forward(agent, sim, action_names, dir_length, frame_idx, id_to_label, target_semantic_id, ob
    while dir_length > 0:
        navigate_and_see(sim, 'move_forward', action_names, frame_idx, id_to_label, target_semantic_i
        dir_length -= forward_length
        frame_idx += 1

    return frame_idx

def navigation(path_3d, obj_points_3d, obj, target_semantic_id):
    sim, cfg = setup_simulation(sim_settings)
    agent = setup_agent(sim, sim_settings, path_3d[-1])

    action_names =
    list(cfg.agents[sim_settings["default_agent"]].action_space.keys())
    print("Discrete action space:", action_names)

    id_to_label =
    load_semantic_annotations('replica_v1/apartment_0/habitat/info_semantic.json')

    if not os.path.exists(f"{obj}_path"):
        os.mkdir(f"{obj}_path")

    frame_idx = 0
    for n in reversed(range(len(path_3d))):
        sensor_state = agent.get_state().sensor_states['color_sensor']
        target_position = obj_points_3d if n == 0 else path_3d[n-1]
        direction, dir_length =
        calculate_direction(sensor_state, target_position)

        frame_idx = rotate_agent(
        agent,
        direction,
        sim,
        action_names,
        frame_idx,
        id_to_label,
        target_semantic_id,
        obj)

        print(f'Forward: {dir_length}')
        frame_idx = move_forward(
        agent,
        sim,
        action_names,
        dir_length,
        frame_idx,
        id_to_label,
        target_semantic_id,
        obj)
```

1. **Initialization:**

- **Simulator setup:** `setup_simulation(sim_settings)` initializes the Habitat simulator with the given settings.
- **Agent setup:** `setup_agent(sim, sim_settings, path_3d[-1])` initializes the agent at the final point in `path_3d`, setting the agent's initial position in the 3D environment.
- **Action space:** Extracts the discrete action space (like `move_forward`, `turn_left`, `turn_right`) for the agent from the simulator configuration (`cfg`).
- **Load semantic annotations:** Loads semantic annotations from a JSON file using `load_semantic_annotations()`, mapping semantic labels to object IDs.
- **Create output directory:** If a directory doesn't exist for saving images for this object (`obj`), it creates one.

2. **Main Loop (Navigating Path):**
   - The function iterates over the 3D path in reverse order (from the final point to the starting point) to navigate the agent towards the target.
   - For each point:
     - **Get agent's sensor state:** The agent's sensor state is retrieved to compute the current position and orientation of the camera sensor.
     - **Target selection:** The next target position is either the object's final location (`obj_points_3d`) or the next point in `path_3d`.
     - **Calculate direction and distance:** `calculate_direction()` computes the normalized direction vector and the distance between the agent's current position and the target.
3. **Rotation:**
   - The agent needs to face the direction of the target. The `rotate_agent()` function:
     - Uses `rotate()` to calculate the yaw difference between the agent's current orientation and the target direction.
     - Iteratively turns the agent (`turn_left` or `turn_right`) until the yaw difference is within an acceptable range, while calling `navigate_and_see()` to capture the RGB image at each frame.
     - Updates `frame_idx` to track the frames saved during rotation.
4. **Forward Movement:**
   - Once the agent is facing the target direction, the `move_forward()` function:
     - Moves the agent forward in small steps (`move_forward` action) until the distance to the target (`dir_length`) is reduced to zero.
     - Captures images along the way using `navigate_and_see()` and increments `frame_idx` for each frame.

Supporting Functions:

- `rotate()` computes the yaw difference between the agent's current orientation and the target direction, returning the required rotation direction (`turn_left` or `turn_right`) and the degree difference.

```
def rotate(agent, direction):
    sensor_state = agent.get_state().sensor_states['color_sensor']
    cur_yaw = math.atan2(2.0 * (sensor_state.rotation.w * sensor_state.rotation.y + sensor_state.r
                         1.0 - 2.0 * (sensor_state.rotation.y**2 + sensor_state.rotation.z**2))
    target_yaw = -math.atan2(direction[0], -direction[1])
    yaw_diff = (target_yaw - cur_yaw + math.pi) % (2 * math.pi) - math.pi
    deg_diff = np.degrees(yaw_diff)
    action = 'turn_left' if deg_diff > 0 else 'turn_right'
    return abs(deg_diff), action
```

  The formula `math.atan2(2.0 * (w * y + x * z), 1.0 - 2.0 * (y^2 + z^2))` is part of the conversion of quaternion to yaw (Euler angle) and is derived from quaternion-to-Euler conversions, getting the current yaw

  After getting the target yaw, calculates the angular difference (`yaw_diff`) between the current yaw (`cur_yaw`) and the target yaw (`target_yaw`).

  To ensure the yaw difference falls within the range of -π to π radians (which ensures the shortest possible rotation), the calculation is normalized using modulo arithmetic:

  - `(target_yaw - cur_yaw + math.pi) % (2 * math.pi)` ensures the difference is wrapped to the range `[0, 2π]`.
  - Subtracting `π` shifts the result back into the range `[-π, π]`, representing the shortest angular difference.

- `move_forward()` iterates the forward movement, reducing the distance step by step, capturing images along the way.

```
def move_forward(agent, sim, action_names, dir_length, frame_idx, id_to_label, target_semantic_id,
    while dir_length > 0:
        navigate_and_see(sim, 'move_forward', action_names, frame_idx, id_to_label, target_semantic
        dir_length -= forward_length
        frame_idx += 1

    return frame_idx
```

If moving ofrward is needed, it uses navigate_and_see to produce each frame, and increase frame index

- `navigate_and_see()` captures the agent's view at each step (forward or turn) and saves RGB images, highlighting regions corresponding to the target object in red based on semantic segmentation.

```
def navigate_and_see(sim, action, action_names, frame_idx, id_to_label, target_semantic_id, obj):
    if action not in action_names:
        return

    observations = sim.step(action)
    rgb_img = transform_rgb_bgr(observations["color_sensor"])

    semantic_id = id_to_label[observations["semantic_sensor"]]
    target_id_region = np.where(semantic_id == target_semantic_id)

    if target_id_region[0].size > 0:
        rgb_img[target_id_region] = cv2.addWeighted(
            rgb_img[target_id_region], 0.5, np.full_like(rgb_img[target_id_region], [0, 0, 255], d
        )

    output_path = os.path.join(f"{obj}_path", f"RGB_{frame_idx}.jpg")
    cv2.imwrite(output_path, rgb_img)
```

I used `cv2.addWeighted` to combine the target object region and the original frame. `np.full_like` is used to compute the region and give the reg filter on the object region. The frames are saved to produce .gif file later.

b. Result and Discussion

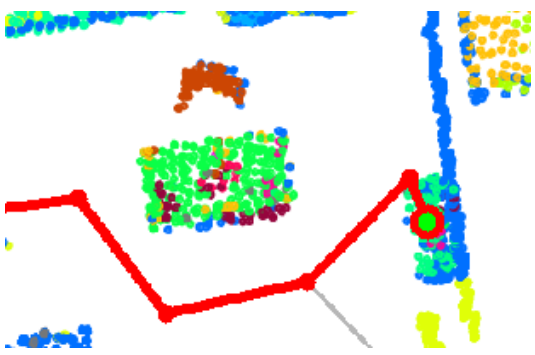
- Show and discuss the results from the RRT algorithm with different start points and targets

  Take rack as example item:

  - Different target points

    As mentioned above, if I set the target points manually, of course it can control the end of path well

But if the target points are not set, then It will calculate the acerage coordinate of the item region, and take it as the target point. Then there is chance the final position is not in front of the item:

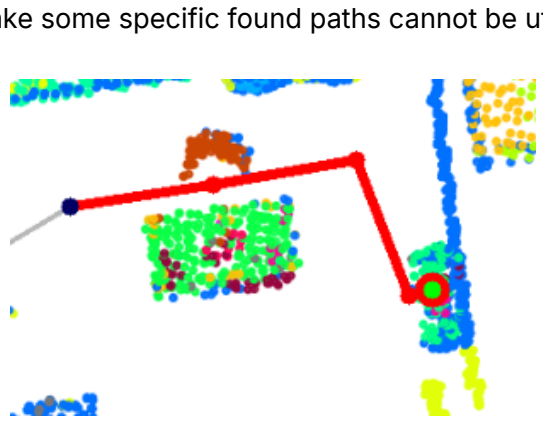


It is because RRT only cares if the target is found or not, but how the camera following the path will face the found target.

- Different start points

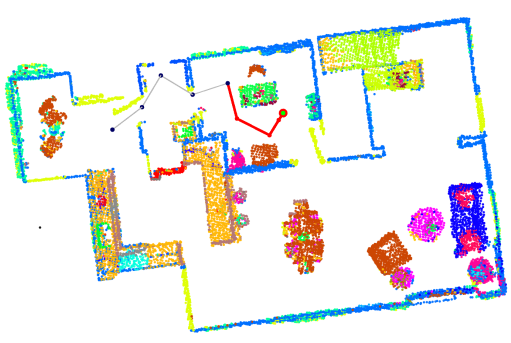
  Different start points make different impacts on the path, and it is related to why I set target points manually.

  So the other reason I decided to set the target points manually is because it helps avoiding uncontrolable issue. For example, the lack of point cloud of some objects make some specific found paths cannot be utilized in navigation:



If I choose the start point right here, rrt will find a path lying between the chair and the desk. But in navigation, the camera cannot pass here. It will just get stuck here.

But if I can set the target point that keeps a distance between the rack, the found path will usually go below (on bev map) the desk:



To conclude, I think setting target points for each item manually is not the best method, but since I did not find a regular pattern to decide each target points and also to avoid the problems above, I think this is so far the best solution for me.

- Discuss about the robot navigation results

  As mentioned on the last question, due to the lack of points of the environment, sometimes the navigation gets stuck.

  The best cases to reach by navigation is the ones which only have one access entry to stay in front of them, ex. stair, cooktop.

And for sofa, cushion, rack, The paths to access them are more complicated because there is things blocking the paths :

c. Reference

- https://blog.csdn.net/feriman/article/details/113775084
- https://blog.csdn.net/You_are_my_dream/article/details/53493752
- https://theclassytim.medium.com/robotic-path-planning-rrt-and-rrt-212319121378

Questions (20%)

a. In the RRT algorithm, you can adjust the step size and bias (the number balance between exploration and exploitation). Please explain how the two numbers affect the RRT sampling result. (15%)

1. Step Size:

The **step size** determines how far the algorithm extends the tree toward a randomly sampled point during each iteration. It influences the granularity and the speed of the tree's growth.

- **Large Step Size:**
  - The tree expands more quickly if the step size is large, covering greater distances in each iteration.
  - **Pros:**
    - The algorithm can reach the goal more quickly, especially in open spaces with few obstacles.
    - It grows the tree much faster.
  - **Cons:**
    - A larger step size increases the risk of missing narrow passages or navigating around complex obstacles, potentially leading to more collisions, which may waste more time.
    - It may result in less detailed exploration of the environment, possibly skipping over fine-grained details of the terrain.
    - There is also chance that it cannot find the target at all because the start point is very close to the target.
- **Small Step Size:**
  - The tree grows more incrementally, with shorter connections between nodes.
  - **Pros:**
    - It allows more precise exploration, particularly in environments with narrow corridors or dense obstacles.
    - A smaller step size can help the algorithm better navigate through complex environments.
  - **Cons:**
    - The tree expands slowly.
    - In open spaces, it may take longer to find a path, as the algorithm may perform unnecessary exploration of areas with no obstacles.

2. Bias (Exploration vs. Exploitation):

Exploitation: How often the algorithm tries to **directly connect to the goal**

Exploration: How often the algorithm tries **randomly sampling points** in the space.

- **High Bias Towards Exploitation (Focus on the Goal):**
  - The algorithm tries to sample points near or directly towards the goal more frequently
  - **Pros:**
    - Faster convergence to the goal if there are few obstacles on the direction toward target.
    - Find a solution quickly, especially in simple environments.
  - **Cons:**
    - It may lead to the algorithm getting stuck in local minima or dead-ends, especially in environments with many obstacles.
    - Less exploration of the environment might mean missing alternative, potentially better paths or exits from complex obstacle zones.
- **Low Bias Towards Exploitation (Focus on Exploration):**
  - The algorithm samples random points in the environment more frequently, with only occasional bias towards the goal
  - **Pros:**
    - Better for environments with complex obstacles.
    - The algorithm has a higher chance of finding paths through difficult-to-reach areas, such as narrow corridors or areas with many obstacles.
  - **Cons:**
    - In simple environments, it may slow down the search, as the algorithm explores unnecessarily when a direct path to the goal exists.

- Can increase computational effort and time due to additional exploration.

b. If you want to apply the indoor navigation pipeline in the real world, what problems may you encounter? (5%)

1. **Sensor Noise and Inaccuracies:** Real-world sensors are prone to noise, inaccuracies, and drift over time. It may result in misinterpretation of the environment, or incorrect mapping. Then the agent may collide with obstacles or failing to correctly identify its position.
2. **Dynamic Environments:** The real-world environment is dynamic. Moving people, animals, furniture will change the structure of the map, so static pre-built maps may not be sufficient to handle changes in the environment. An agent relying on static maps could easily get lost if the environment changes unexpectedly.
3. **Real-Time Performance:** Algorithms that work well in simulation may not perform quickly enough in the real world, especially in computationally constrained platforms. And the slow decision-making could lead to delays in movement or the inability to react quickly to dynamic obstacles, causing navigation failures.