49274 Space Robotics

Assignment 2: Path Planning for a Planetary Rover

Introduction:

We implement a path planning system. First with naïve planner then move to A*. we see how energy cost inclusion affects the path and finally we see the PRM method.

Reflection:

Most of the time was spend in task 5. First it got stuck in the infinite loop when implemented so I changed the beta value from 0.5 to 0.3. then it worked fine once changed to 0.3 and it converged quicker.

Video Link:

https://drive.google.com/file/d/1AspiGKCvw9G0WcMONKYi4l3MIoyoOmDN/view?usp=sharing

Task 1:

In Task 1, we loop through all the nodes in self.nodes_, where each node has coordinates (x, y) and an index (idx). For each node, we calculate the Euclidean distance between the node and the input point xy. It returns the index of the node that is closest to the input point.

```
        ## Task 1          ##
        ####################
        low_dist = math.sqrt((self.nodes_[i].x - xy[0]) ** 2 + (self.nodes_[i].y - xy[1]) ** 2)
        if low_dist < best_dist:
            best_dist = low_dist
            best_index = self.nodes_[i].idx

    return best_index
```

```python
        best_dist = 999999999
        best_index = None

        for i in range(len(self.nodes_)):

            ####################
            ## YOUR CODE HERE ##
            ## Task 1         ##
            ####################
            low_dist = math.sqrt((self.nodes_[i].x - xy[0]) ** 2 + (self.nodes_[i].y - xy[1]) ** 2)
            if low_dist < best_dist:
                best_dist = low_dist
                best_index = self.nodes_[i].idx

        return best_index
```

Task 2:

The requirement for this naïve planner is to start at the current node, then pick the next neighbouring node that has not yet been visited and is closest to the goal. If the path gets stuck, then it is allowed to revisit a node up to 3 times.

In the naive_path_planner, the method starts at the given start node and iteratively selects the best neighbor based on the Euclidean distance to the goal node while applying a penalty for revisiting neighbors multiple times. Inside the loop, it checks all neighboring nodes, calculates their distance to the goal (with the penalty), and updates the current node to the best neighbor. This process continues until the goal node is reached or no valid neighbors are found. The method returns the final path as a list of nodes from the start to the goal.

```python
def naive_path_planner(self, start_idx, goal_idx):

    ####################
    ## YOUR CODE HERE ##
    ## Task 2         ##
    ####################
    path = []

    current = self.graph_.nodes_[start_idx]
    path.append(current)

    while current.idx != goal_idx:
        best_neighbour = None
        best_neighbour_distance = float('inf')
        for neighbour in current.neighbours:
            count_visits = path.count(neighbour)
            if count_visits < 3:
                distance_to_goal = neighbour.distance_to(self.graph_.nodes_[goal_idx]) + 100 * count_visits
                if distance_to_goal < best_neighbour_distance:
                    best_neighbour = neighbour
                    best_neighbour_distance = distance_to_goal
        if best_neighbour is None:
            # If no valid neighbours, exit
            break
        else:
            # Otherwise, move to the best neighbour
            current = best_neighbour
            path.append(current)

    return path
```

Task 3:

```python
## Task 3         ##
####################
# Select a node with the minimum cost
min_node_index_in_unvisited = self.get_minimum_cost_node(unvisited_set)
node_idx = unvisited_set[min_node_index_in_unvisited]
```

```python
# Move the node to the visited set

visited_set.append(node_idx)
unvisited_set.remove(node_idx)
```

```python
# Termination criteria
# Finish early (i.e. "return") if the goal is found

if node_idx == goal_idx:
    rospy.loginfo("Goal found!")
    return
```

```python
else:
    # Compute the cost of this neighbour node
    # hint: cost_to_node = cost-of-previous-node + cost-of-edge
    # hint: cost_to_node_to_goal_heuristic = cost_to_node + self.heuristic_weight_ * A*-heuristic-score
    # hint: neighbour.distance_to() function is likely to be helpful for the heuristic-score

    new_cost_to_node = self.graph_.nodes_[node_idx].cost_to_node + neighbour_edge_cost
    new_cost_to_goal_heuristic = new_cost_to_node + self.heuristic_weight_ * neighbour.distance_to(
        self.graph_.nodes_[goal_idx])
```

```python
# Check if neighbours is already in unvisited
if neighbour.idx in unvisited_set:

    # If the cost is lower than the previous cost for this node
    # Then update it to the new cost
    # Also, update the parent pointer to point to the new parent

    if new_cost_to_goal_heuristic < neighbour.cost_to_node_to_goal_heuristic:
        neighbour.parent_node = self.graph_.nodes_[node_idx]
        neighbour.cost_to_node = new_cost_to_node
        neighbour.cost_to_node_to_goal_heuristic = new_cost_to_goal_heuristic
```

```python
            else:
                # Add it to the unvisited set
                unvisited_set.append(neighbour.idx)
                # Initialise the cost and the parent pointer
                # hint: this will be similar to your answer above
                neighbour.parent_node = self.graph_.nodes_[node_idx]
                neighbour.cost_to_node = new_cost_to_node
                neighbour.cost_to_node_to_goal_heuristic = new_cost_to_goal_heuristic

    # Visualise the current search status in RVIZ
    self.visualise_search(visited_set, unvisited_set, start_idx, goal_idx)

    # Sleep for a little bit, to make the visualisation clearer
    rospy.sleep(0.01)
```

In Task 3, the search method begins by initializing all nodes with large costs and no parent nodes. The unvisited and visited sets are also initialized, with the start node added to the unvisited set. The method then enters a loop where the node with the minimum cost from the unvisited set is selected and moved to the visited set. If the selected node is the goal node, the search terminates. The loop continues until the goal is found or there are no more unvisited nodes.

For each neighbor of the current node, the method checks if the neighbor is already visited. If the neighbor is in the visited set, it is skipped. If not, the cost to reach the neighbor is calculated based on the current node's cost and the edge cost.

cost_to_node = cost-of-previous-node + cost-of-edge

*cost_to_node_to_goal_heuristic = cost_to_node + self.heuristic_weight_ * A*-heuristic-score*

If the neighbor is already in the unvisited set, its cost is compared with the new cost, and if the new cost is lower, the neighbor's cost and parent node are updated. If the neighbor is not in the unvisited set, it is added, and its parent and cost are initialized.

Task 4:

Here we reconstruct the path from the goal node back to the start node. It starts by initializing the path with the goal node and then follows the *parent_node* attribute of each node, appending each parent node to the path. This process continues until the start node is reached (when the *parent_node* is None). Finally, the path is reversed to present it in the correct order from start to goal.

```
    def generate_path(self, goal_idx):
        # Generate the path by following the parents from the goal back to the start
        path = []
        current = self.graph_.nodes_[goal_idx]
        path.append(current)
        ####################
        ## YOUR CODE HERE ##
        ## Task 4         ##
        ####################
        while current.parent_node is not None:
            current = current.parent_node
            path.append(current)
        path.reverse()

        return path
```

Task 5:

In this task, the smooth waypoints are first initialized as the original waypoints. Then enters a loop that continues until the total change between iterations (difference) is smaller than the specified tolerance (0.001). Within the loop, for each waypoint except the first and last, the x and y coordinates are updated using a formula that incorporates the current and neighboring points, scaled by alpha and beta. After each update, the path is checked for collisions using the *is_occluded* function to ensure no path segment intersects with obstacles. If a collision is detected, the update is discarded, and the waypoint reverts to its original value. Finally, the total difference between the original and new path points is calculated, and the process repeats until the path changes minimally, at which point the smoothed path is returned.

```python
        ## Task 5        ##
        ####################
        tolerance = 0.001
        difference = tolerance
        while difference >= tolerance:
            difference = 0.0
            path_smooth_new = copy.deepcopy(path_smooth)
            for i in range(1, len(path_smooth) - 1):
                # Update the smooth path using the formula
                path_smooth_new[i].x = path_smooth[i].x - ((alpha + 2 * beta) * path_smooth[i].x) + \
                                       (alpha * path[i].x) + (beta * path_smooth[i - 1].x) + (
                                               beta * path_smooth[i + 1].x)
                path_smooth_new[i].y = path_smooth[i].y - ((alpha + 2 * beta) * path_smooth[i].y) + \
                                       (alpha * path[i].y) + (beta * path_smooth[i - 1].y) + (
                                               beta * path_smooth[i + 1].y)
                if is_occluded(self.graph_.map_.obstacle_map_, [path_smooth_new[i - 1].x, path_smooth_new[i - 1].y],
                               [path_smooth_new[i].x, path_smooth_new[i].y]) \
                        or is_occluded(self.graph_.map_.obstacle_map_, [path_smooth_new[i].x, path_smooth_new[i].y],
                                       [path_smooth_new[i + 1].x, path_smooth_new[i + 1].y]):
                    path_smooth_new[i] = copy.deepcopy(path_smooth[i])
            for i in range(len(path_smooth)):
                difference += (path_smooth_new[i].x - path_smooth[i].x) ** 2 + (
                        path_smooth_new[i].y - path_smooth[i].y) ** 2
            path_smooth = copy.deepcopy(path_smooth_new)

        return path_smooth
```

Task 6:

energy consumption is incorporated into the edge costs between nodes to favor flat or downhill routes over steep uphill paths. For each edge, the 2D pixel coordinates of the two nodes are converted into 3D world coordinates using the pixel_to_world function. The 3D Euclidean distance (dx) and the change in elevation (delta_z) are computed, and the slope angle (θ) is calculated using the arctangent of the elevation change over the horizontal distance. The energy cost E is then calculated based on the slope, gravity, mass, and rolling resistance, with the absolute value ensuring all costs remain positive.

u = 0.1 # Rolling resistance coefficient
m = 1025 # Mass of the rover in kg
g = 3.71 # Gravity on Mars in m/s^2

If energy costs are enabled, this energy cost is assigned to the edge; otherwise, the edge cost defaults to the 2D Euclidean distance.

```python
if self.use_energy_costs_:
    # Set the edge costs as estimated energy consumption
    ###################
    ## YOUR CODE HERE ##
    ## TASK 6        ##
    ###################
    # energy_cost = distance # Comment this out once you've done this Task
    [x1, y1, z1] = self.map_.pixel_to_world(node_i.x, node_i.y)
    [x2, y2, z2] = self.map_.pixel_to_world(node_j.x, node_j.y)

    dx = math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2 + (z2 - z1) ** 2)
    delta_z = z2 - z1
    horizontal_distance = math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)

    theta = math.atan2(delta_z, horizontal_distance)

    E = abs((u * m * g * math.cos(theta) + m * g * math.sin(theta)) * dx)

    energy_cost = E  # Set the energy cost for this edge

else:
    # Define the edge cost as standard 2D Euclidean distance in pixel coordinates
    energy_cost = distance
# Create the edge
node_i.neighbours.append(node_j)
node_i.neighbour_costs.append(energy_cost)
```

Task 7:

The goal is to create a Probabilistic Roadmap (PRM) by randomly generating a specified number of nodes (num_nodes) in the free space of the environment. The code enters a loop that continues until the required number of nodes is generated. For each iteration, random x and y coordinates are selected within the bounds of the environment using random.randint(). The code then checks whether the selected location is free (not occupied by obstacles) using self.map_.is_occupied(x, y). If the location is free, a new node is created and added to the node list (self.nodes_), and the index is incremented. This process ensures that nodes are placed only in the free space, just like the node generation process in a grid.

```python
        ## Task 7          ##
        ####################
        while idx < num_nodes:
            #x = random.randint(self.map_.min_x_, self.map_.max_x_)
            #y = random.randint(self.map_.min_y_, self.map_.max_y_)
            x = int(np.random.uniform(self.map_.min_x_, self.map_.max_x_))
            y = int(np.random.uniform(self.map_.min_y_, self.map_.max_y_))
            if rospy.is_shutdown():
                return

            occupied = self.map_.is_occupied(x, y)

            if not occupied:
                self.nodes_.append(Node(x, y, idx))
                idx += 1
```

```python
## TASK 6 -- after TASK 7 ##
############################
# energy_cost = distance # Comment this out once you've done this Task
# energy_cost = ??
[x1, y1, z1] = self.map_.pixel_to_world(node_i.x, node_i.y)
[x2, y2, z2] = self.map_.pixel_to_world(node_j.x, node_j.y)

dx = math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2 + (z2 - z1) ** 2)
delta_z = z2 - z1
horizontal_distance = math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)

theta = math.atan2(delta_z, horizontal_distance)

E = abs((u * m * g * math.cos(theta) + m * g * math.sin(theta)) * dx)

energy_cost = E  # Set the energy cost for this edge
```

Task 8:

I tried to implement this, and it works in separating the nodes group. But it couldn't separate the nodes when the prm_max_edge_length is changed to 50. It still shows as a single node group.

```python
## Task 8          ##
####################
graph_search_obj = GraphSearch(self)

groups = [-1] * len(self.nodes_)  # -1 indicates ungrouped nodes

group_number = 0

for i in range(len(self.nodes_)):
    if groups[i] == -1:
        connected_nodes = graph_search_obj.find_connected_nodes(i)

        print(f"Group {group_number + 1}: Nodes connected to node {i} -> {connected_nodes}")

        group_number += 1

        for node in connected_nodes:
            groups[node] = group_number

# Save the groups so they will show up in visualization
self.groups_ = groups
```

```python
for n in self.graph_.nodes_:
    n.cost_to_node = float('inf')  # A large number to represent "unvisited"
    n.parent_node = None  # Reset parent to None

    # Setup unvisited and visited sets
unvisited_set = []
visited_set = []

# Add the start node to the unvisited set
unvisited_set.append(start_idx)
self.graph_.nodes_[start_idx].cost_to_node = 0  # Cost to reach start node is 0

# Loop until there are no more nodes in the unvisited set
while len(unvisited_set) > 0:

    # Select the node with the minimum cost (Dijkstra's logic for selecting the closest node)
    min_node_index_in_unvisited = self.get_minimum_cost_node(unvisited_set)
    if min_node_index_in_unvisited is None:  # If no valid node found, exit loop
        break
    node_idx = unvisited_set[min_node_index_in_unvisited]

    # Move the selected node to the visited set
    visited_set.append(node_idx)
    unvisited_set.remove(node_idx)

    # Explore neighbors of the current node
    for neighbour_idx in range(len(self.graph_.nodes_[node_idx].neighbours)):
        neighbour = self.graph_.nodes_[node_idx].neighbours[neighbour_idx]
        neighbour_edge_cost = self.graph_.nodes_[node_idx].neighbour_costs[neighbour_idx]

        # If the neighbor is already visited, skip it
        if neighbour.idx in visited_set:
            continue
```

```
                # Calculate the new cost to reach the neighbor
                new_cost_to_node = self.graph_.nodes_[node_idx].cost_to_node + neighbour_edge_cost

                # If the neighbor is already in unvisited, check if the new cost is lower
                if neighbour.idx in unvisited_set:
                    if new_cost_to_node < neighbour.cost_to_node:
                        # Update the cost and parent node
                        neighbour.cost_to_node = new_cost_to_node
                        neighbour.parent_node = self.graph_.nodes_[node_idx]
                else:
                    # If neighbor is not in the unvisited set, add it and update its cost and parent
                    unvisited_set.append(neighbour.idx)
                    neighbour.cost_to_node = new_cost_to_node
                    neighbour.parent_node = self.graph_.nodes_[node_idx]

        # Return all visited nodes as a list (i.e., all nodes connected to start_idx)
        return visited_set
```

Conclusion:

If I had more time I could have debugged the task 8 and made it working with separating the node groups.