Motion-Planning HW3

Nir Manor 305229627, Ortal Cohen 308524875

07/04/2024

1 Robot Modelling

1.1 Compute Distance

In this function, we aim to compute the Euclidean distance between two given configurations of the manipulator robot. We start by computing the forward kinematics for both configurations using the `compute forward kinematics` function. Then, we calculate the Euclidean distance between the resulting coordinates of the two configurations using numpy's `linalg.norm` function.

1.2 Compute Forward Kinematics

This function computes the forward kinematics of the manipulator robot given a configuration. It calculates the 2D position (x, y) of each link, including the end-effector, and returns it as a numpy array. The function iterates through each link in the given configuration, computes the angle of the link relative to the base, and then calculates the x and y positions using trigonometric functions. It updates the base position for the next link and appends the position of each link to the position vector. Finally, it returns the position vector as a numpy array.

1.3 Validate Robot

We found that the simplest way to check if a configuration would contain self-collisions is to build a LineString object (from the shapely library) from the given link positions, and then to use the `is_simple` attribute of the LineString class to check if the robot's geometry is simple, meaning it contains no self-collisions anywhere (except maybe the boundary points, in this case the origin and the end-effector). The `is_simple` attribute evaluated to True if the object is indeed simple.

2 Motion Planning

2.1 Performance Metrics

The performance metrics for different scenarios are summarized in Table 1. Additionally, the performance for goal biasing of 5% and 20% (with E2 and E1), averaged over 10 executions, is as follows:

2.2 Table: Cost and Time for Each Scenario

	E1-0.05		E1-0.2		E2-0.05		E2-0.2	
	Cost	Time	Cost	Time	Cost	Time	Cost	Time
1	213.16	0.21	181.62	0.3	210.13	11.32	232.39	11.71
2	232.07	9.9	212.86	83.33	224.06	14.81	265.25	1.65
3	249.87	1.08	220.52	2.06	227.16	0.75	278.15	13.26
4	277.76	0.32	233.38	70.53	230.01	4.39	287.23	1.47
5	315.37	9.1	242.43	28.11	263.75	5.14	288.21	14.36
6	341.72	6.02	318.63	82.23	266.01	17.43	349.08	36.22
7	405.57	37.16	319.1	18.35	293.54	6.01	362.34	12.63
8	439.43	19.63	413.78	17.03	351.83	13.29	392.9	8.76
9	485.85	70.46	480	20.88	526.85	70.01	446.59	70.06
10	486.14	51.27	500.19	37.38	594.45	41.01	575.07	69.75
Average Cost	344.694		312.251		318.779		347.721	
Average Time	20.515		36.02		18.416		23.987	

Table 1: Cost and Time for Each Scenario

Inspection Planning with RRT

In the inspection planning task, we adapted the RRT algorithm to efficiently explore and inspect a set of points of interest (POI) without the constraint of reaching a predefined goal. Here's how we implemented the functions:

Sampling Random Configuration

This process selects potential robot configurations randomly within the configuration space. In traditional RRT, configurations are uniformly sampled with occasional bias towards the goal state. However, in inspection planning there is no goal configuration and therefore the sampling strategy prioritizes unexplored regions containing new inspection points. We maintain a list of all inspection points seen so far, allowing us to check configurations that will bring new inspection points to the tree.

Adding Inspection Points

Compute Union of Points

Implemented in `MapEnvironment.py`, the `compute_union_of_points` function computes the union of two sets of inspection points, crucial for tracking the inspection progress along the tree.

Compute Inspection Points between 2 Configuration

When adding a new vertex to the tree we realized that we need to consider all the inspection points between the existing configuration (father) to the new configuration (child) that is being added. The set of inspection points of the new vertex is set to the union between the inspection points of the father and the inspection points along the path between father to child.

Plan Function and Rewiring Option

The main planning algorithm is encapsulated within the plan function. It initializes the plan, sets up the starting configuration, and iteratively extends the tree until the desired coverage threshold is reached. Additionally, we introduced rewiring to optimize the inspection process in certain cases. This decision is based on comparing the coverage of all inspection points observed so far across the entire tree to the configuration that saw the most inspection points. If there's a significant gap between them, indicating potential improved coverage, we initiate the rewiring process.

Rewiring Hyperparameters

After several trial-and-error experiments, we determined two key hyperparameters for the rewiring process. Firstly, we set the number of nearest neighbors (*k-nearest neighbors*) to 5. This value represents the number of nearby vertices that we consider when evaluating the rewiring option. A higher value may lead to increased computational complexity, while a lower value may overlook potentially beneficial rewiring opportunities.

Secondly, we established a condition for deciding when to initiate the rewiring process. Specifically, we compare 1.8 times the number of inspection points of the vertex with the maximum inspection points to the union of all inspection points observed in the tree. If this threshold is met, indicating a significant discrepancy between the potential coverage of a new configuration and the current coverage of the tree, we trigger the rewiring process. Additionally, if all inspection points in the environment are included in the union of inspection points across all vertices in the tree, we also initiate

rewiring. This condition ensures that even if all points are observed, the planner can attempt to reassemble the tree to potentially discover new paths or improve coverage efficiency.

Propagate To Children

Propagates changes in inspection points and costs to children vertices after rewiring.

3 Algorithm Performance

From Table 2, it can be observed that for a coverage of 0.5, the average cost of the path was 364.079 units, with an average execution time of 34.184 seconds. On the other hand, for a coverage of 0.75, the average cost increased to 456.085 units, with a longer average execution time of 106.552 seconds. These results indicate that as the coverage increases, the algorithm tends to explore a larger portion of the configuration space, resulting in a higher cost path and longer execution time.

	Coverage 0.5		Coverage 0.75	
2*	Cost	Time	Cost	Time
1	281.28	25.39	360.8	79.3
2	317.71	22.65	368.62	36.55
3	333.05	56.67	379.31	86.18
4	343.06	12.39	387.52	204.02
5	350.73	66.92	405.44	156.79
6	370.97	34.67	435.05	45.49
7	397.7	20.96	469.53	43.3
8	399.01	62.04	488.06	80.82
9	420.79	12.15	610.92	232.54
10	426.49	28	655.6	100.53
Average Cost	364.079		456.085	
Average Time	34.184		106.552	

Table 2: Cost and Time for Each Scenario