Algorithmic Robot Planning
Homework 3

Andrew Elashkin, Yonatan Sommer

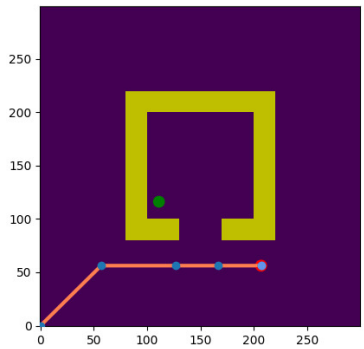
March 2022

1 Robot Modelling

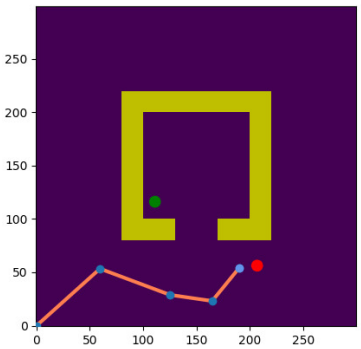
- compute_distance: Using Numpy functions, we calculated the euclidean distance by definition.
- compute_forward_kinematics - In this method we are required to translate the angles of the links to x, y coordinates of each link's ends. We do this iteratively by calculating the each link vertex (end) using the previous vertex's coordinates and angle relative to the x axis (with $(0, 0)$ as the first coordinates). We do this as follows using basic trigonometry: Let the previous link vertex be at coordinates (x_0, y_0) and it's angle θ_0.
  1. Compute the angle θ_1 of the next link (relative to the x axis) using the method compute_link_angle
  2. $x_1 = x_0 + \cos\theta_1$
  3. $y_1 = y_0 + \sin\theta_1$
- validate_robot - using the Shapely method Linestring.intersects and Shapely.overlap we asses whether any link colllides with another. The last check is whether the last two links completely overlap, which occcurs when they both have identical coordinates. This is necessary since Linestring.overlaps returns false if two Linestrings completely overlap, therefor for the last two links which are the same size we must do this extra check

2 Motion Planning

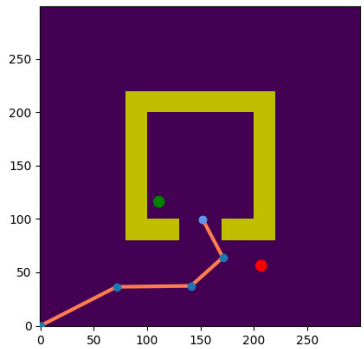
In our implementation we assumed that the goal is reachable, i.e. there exists some path from the start to the goal. If provided map requires unreachable path our implementation will stuck in infinite loop. We based our implementation on the previous homework on motion planning for the robot with 2 degrees of freedom on 2D map. Since our code was initially generic we only had to replace 2D coordinates with configurations to make it work in this new setting.



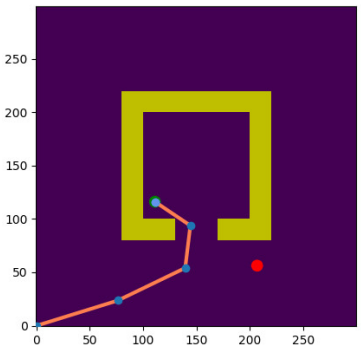
(a) Step 1



(b) Step 2



(a) Step 3



(b) Step 4

Figure 2: Motion planning task run simulation for RRT E1 with 0.05 goal bias

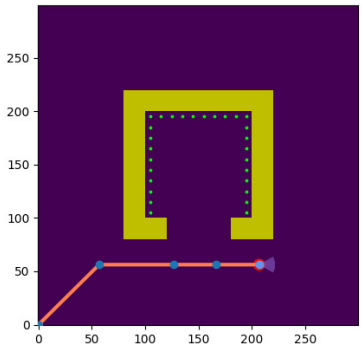
We present the results for the motion planning runs as requested for goal bias 0.05 and 0.2 and in both E1 and E2 modes. For E2 we used a step size of 3.

Mean run time (sec)	Result
Bias 5% E1	68.86
Bias 5% E2	11.21
Bias 20% E1	30.65
Bias 20% E2	9.53

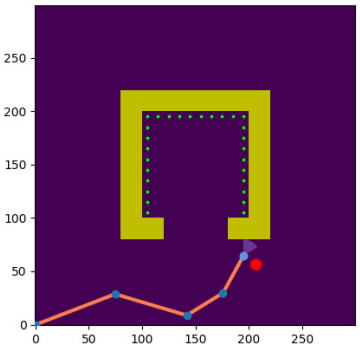
Mean path cost	Result
Bias 5% E1	7.11
Bias 5% E2	6.34
Bias 20% E1	5.97
Bias 20% E2	5.8

Run visualisations for different modes and biases are attached to the submission as GIFs. One run for mode E1 and goal bias 0.05 can be observed on Figure 2. It is worth noting that we observed a high variance in the run times for E1. For example, in the E1 0.05 goal bias runs, there were 4 out of 10 runs that took less then 2 seconds and 2 that took over 150 seconds. This suggests a significant effect of the random sampling on runtime which may be due to the map and the relatively small area we need to fit the robot through. This high variance causes the average runtime results to be less meaningful when averaged over a small sample size.

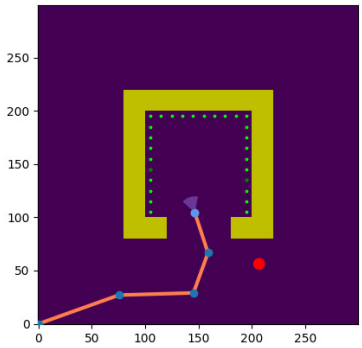
3 Inspection Planning



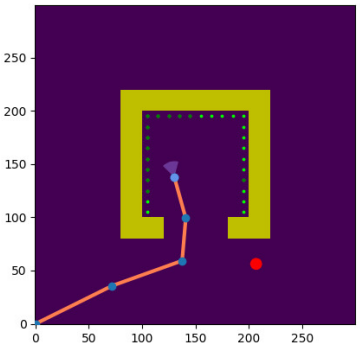
(a) Step 1



(b) Step 2



(a) Step 3



(b) Step 4

Figure 4: Inspection planning RRT run simulation for 0.5 coverage

In inspection planning task, we were required to inspect a set of points of interest (POI). We started by adapt our existing RRT implementation to handle the inspection task. We've changed the stopping condition of our planner from reaching a goal vertex to reaching a vertex in an RRT tree

with total coverage higher than given threshold. By the total coverage for vertex v_c here we mean the sum of the coverages of all vertices on the path from the start node v_s to the v_c. Unfortunately this approach (that we will refer as vanilla) does not converge in reasonable time for the given setting for coverages higher than 0.5.

A table with the results for vanilla can be seen below. Since there is no goal bias we ran the algorithm for E1 and E2 modes only. Some of the runs for coverage 0.75 had to be timed out at 20 minutes, making the mean run time not representative. Such cases are marked in the table.

Mode	Mean run time (sec)	Path cost (coverage 0.5)
E1	151.12	8.0
E2, step = 1	86.88	6.74
E2, step = 0.4	100.41	6.24

Mode	Mean run time (sec)	Path cost (coverage 0.75)
E1	More than 600	5.74 (for converged runs)
E2, step = 1	More than 1200	6.12 (for converged runs)

From the results above we can observe that if vanilla RRT converges it produces path with relatively low cost, however that at minimum takes a lot of computation time or even don't converge at all.

To deal with this problem we have also implemented an improved version of RRT-Inspection algorithm that we will refer to as k-RRTI. The main idea behind k-RTTI is to connect new vertex not to the closest vertex in the tree, but to look at the k-closest vertices and to choose the one with highest coverage from them.

Mode	Mean run time (sec)	Path cost (coverage 0.5)
E1	21.69	16.55
E2, step = 1	47.08	13.54

Mode	Mean run time (sec)	Path cost (coverage 0.75)
E1	60.83	28.45
E2, step = 1	89.50	15.17

We can clearly see that this approach now converges for high coverages and produces results faster than vanilla RRT. For example, we can see from the results of E1 on 0.5 coverage that the running time got reduced by 86%, but the average path length got 2.07 times longer as a result of such trade-off.

Based on our results we assume that the reason vanilla RRT does not converge for high coverages is the presence of multiple goals. RRT is capable of sampling one configuration with high coverage and the path to it, but in the inspection task we want to sample multiple best configurations and have one path through all of them. However, with vanilla RRT new configuration will be connected to the closest vertex in the tree, meaning that samples with high coverage will lay on different branches of the tree and will never be on the same path from the root because of that. The hypothesis we for k-RRT-inspection was that by the way new vertex is connected to the graph we will be able to effectively sample in the regions with high coverage that already had samples, but those samples had paths with low coverage from the root.

Another possible approach to overcome this flaw of RRT will be the introduction of full-fledged

rewiring mechanism, but it is beyond the scope of this work.