Algorithmic Robot Planning
Final Project

Andrew Elashkin, Yonatan Sommer

March 2023

Goal of the project: to provide a motion plan that will maximize the amount of time in which the end-effector of the gripping robot is under the field-of-view of the inspecting robot.

Solution: To solve this problem we've proposed several algorithms described below.

1 Baseline Sampling Based Algorithm

We started by implementing a simple sampling based algorithm:

1. Initialize inspector robot plan with origin in timestamp 0
2. While coverage $<$ goal coverage, do:
   (a) Sample a configuration q for $\mathcal{R}_{\text{inspect}}$. If q is invalid, re-sample.
   (b) Iterate over all unobserved timestamps and for each such timestamp t:
      i If $\mathcal{R}_{\text{inspect}}$ doesn't inspect $\mathcal{R}_{\text{task}}$ from q at timestamp t: go back to step (a).
      ii Denote the previous and next configurations in the current plan relative to t as *previous* and *next*. If one (or both) of the edges *previous-q* or *q-next* is invalid, go back to step (a).
      iii Denote current_coverage as the number of timestamps observed by $\mathcal{R}_{\text{task}}$ during the edge *previous-next* and new_coverage as the total number of unique timestamps observed during the edges *previous-q* and *q-next*. Calculate current_coverage and new_coverage.
      iv if new_coverage $>$ current_coverage, add q to plan at timestamp t.

This algorithm (NaiveTaskInspectionPlanner) isn't complex but have performed quite well on the provided maps. We were able to achieve 0.64% coverage on map1 and 0.97% on map2 and the paths appear pretty smooth and efficient.

If a given map satisfies two conditions:

1. All points that can be inspected have non-point areas from where they can be observed (i.e. infinite number of configurations that can observe them)
2. All valid configurations are collision-free (i.e. we can always have an edge between two valid configurations)

Then this algorithm is probabilistically complete in terms of reaching the max possible coverage (eventually for any uninspected timestamp we will probabilistically sample a configuration that inspects it and we will be able to add it to our plan).

1.1 Improvements

We then attempted to improve on the baseline sampling algorithm by implementing a new version (PubertyTaskInspectionPlanner) that has three modes:

1. Don't Replace Vertex : This is the same as the previously described baseline algorithm - we only add a configuration to the plan when we don't have an existing vertex set there.
2. Always Replace Vertex : If a new sample improves coverage, add even when replacing an existing vertex in the plan.
3. Decreasing vertex Replacement : This is a middle ground of the previous two modes - as we achieve more coverage, the rate we attempt to replace existing vertices decreases. We do this by defining an epsilon that grows with proportion to the achieved coverage and defines a decreasing probability for allowing replacements of existing vertices.

This version was also implemented with various container to store calculations that would be needed later to improve efficiency in computation time.

We expected the trade-off between modes 1 and 2 to be computation time for lower cost plans (mode 2 would take a lot longer due to more complex calculations and many samples that we attempt to add, but we presumed would provide more efficient paths). Mode 3 was meant to balance out the trade off. We were surprised to discover that mode 1 was not only orders of magnitude faster then the other modes getting to the desired coverage, but also got lower cost paths. Furthermore, modes 2 and 3 were barely able to improve on the max coverage we could achieve on the given maps. When investigating the results we observed that the paths provided by the other modes included many "jumps" and were far less smooth then the paths provided by mode 1.

We spent a long time trying to understand the unexpected results. We observed that the paths provided by mode 1 always had a small number of configurations (usually 5-7) and the paths were very simple. Usually the first few "good" configurations would fill the plan and were able to reach high coverage in a short time. By allowing replacement of vertices in modes 2 and 3, "jumps" in the path were created in order to inspect an extra timestamp and once the jumps existed in the plan they were very hard to "smooth out" (you would need to sample a configuration that observes the timestamp and is closer to the other configurations before and after). It turns out the plan would by better by ignoring these "jump" opportunities and reaching the desired coverage without replacement by continuing to sample many points. The replacement of vertices is the computational bottleneck of this algorithm and therefor we spend most of the runtime attempting to insert vertices rather then ample new good points.

1.1.1 Biased Sampling

We also implemented biased sampling to improve intermediate configurations. For each two steps s_1 and s_2 from our plan $\mathcal{P}$ that are not consecutive (i.e. $t_{s2} - t_{s1} > 1$) we try to sample an intermediate point from a Gaussian distribution over the average value of (q_{s1}, q_{s2}) with variable SD. The idea of this approach is to "smooth out" the path in the best possible way to reduce total cost and improve the coverage. We can adjust insertion conditions depending on the kind of optimisation we are looking for by giving bigger weight to coverage or cost when deciding on the new sample inclusion to the plan.

2 RRT with timestamps

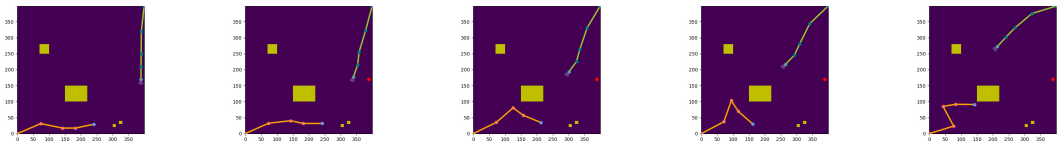
As our next approach we've decided to modify the RRT-Inspection planner from the HW3. In the inspection planning task, we were required to inspect a set of points of interest (POI). We adapted our existing RRT implementation to handle the inspection task by changing the stopping condition of our planner 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. However, this approach does not take into account the order in witch we need to visit those POI's.
To overcome this problem we need to base our sampling not only on configurations, but also on timestamps. The approach is similar to k-RRT that we've proposed in our previous work. We base the vertex connection on the timestamp and then look for the configurations in the previous timestamps with smallest euclidean distance from the sample in that configuration.
First, we sample a random configuration q_{sample} and check what POI from the trajectory τ of R_{task} can be observed from it (regardless of the timestamp). Then, for each step $s_{cur} = \langle q_{cur}, t_{cur} \rangle$ with $q \in X$ denoting a configuration and $t \in R$ denoting the time in τ we check if the q_{sample} is observing q_{cur} and can be connected to one of the $s_{previous} = \langle q, t_{previous} \rangle$ where $t_0 < t_{previous} < t_{cur}$. We stop sampling when the desired coverage is achieved.

3 Experiments

We run our experiments with the setups described in sections 1 and 2 on both maps that were provided to us. As we mentioned before, we got the best results from the most simplistic planner. We think that it is mainly because of the two reasons: First, the maps does not require us to follow close the task robot, there are configurations that guarantee us 0.3 coverage by simply standing there whole run. Second, we are allowed to jump from one configuration to another instantly, meaning that we only need to chain small amount of such points. Because of those reasons we've focused on improving the baseline algorithm and not on RRT based approach. The results below are for the baseline and its improved versions.

3.1 Results for Map 1

Maximum coverage that we were able to achieve for this map and given task was 64%.



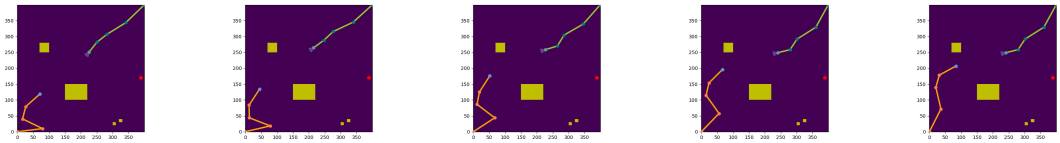
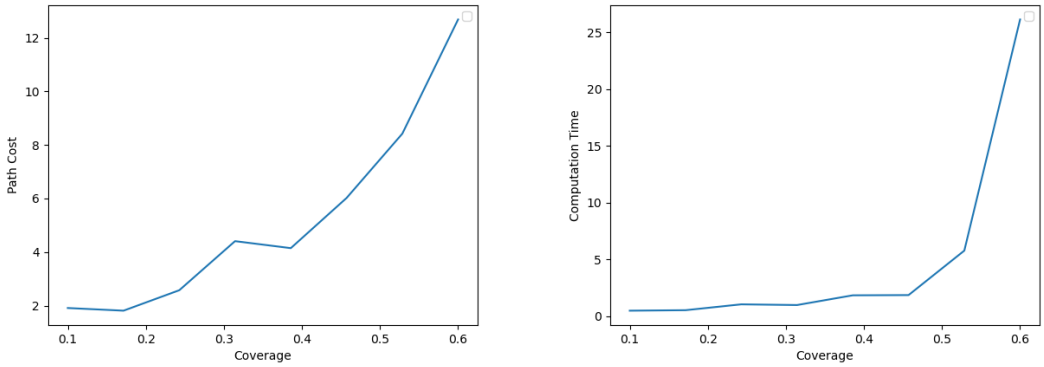


Figure 2: Storyboard for one run with coverage 0.6 on the Map 1



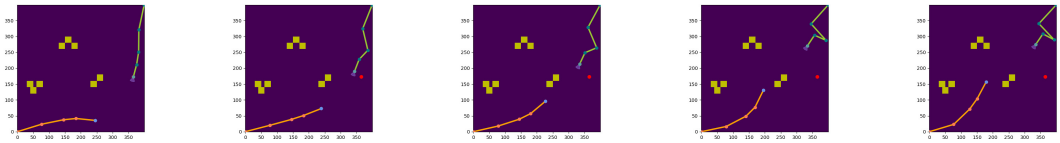
(a) Path cost (radians) / Coverage

(b) Computation Time (sec) / Coverage

Figure 3: Coverage as a function of the computation time / path length, average over 10 runs

3.2 Results for Map 2

Maximum coverage that we were able to achieve for this map and given task was 97%.



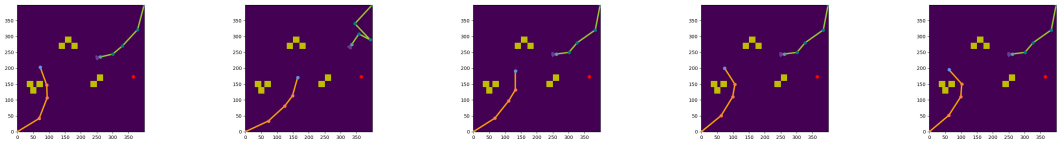
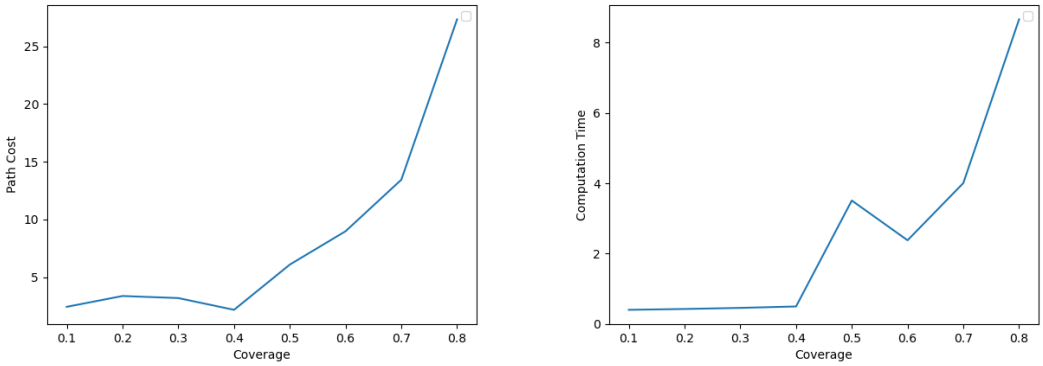


Figure 5: Storyboard for one run with coverage 0.8 on the Map 2



(a) Path cost (radians) / Coverage

(b) Computation Time (sec) / Coverage

Figure 6: Coverage as a function of the computation time / path length, average over 10 runs

3.3 Difference between the modes

To better show the difference between the modes we run it on the high coverage setting. Quick reminder about the modes we've used for Baseline apgorithm:

- M1 - Don't replace
- M2 - Always replace
- M3 - Decreasing replacement

Map 1, coverage 0.5, difference between the modes:

Mode	Mean run time (sec)	Path cost
M1	4.49	7.03
M2	46.62	12.01
M3	20.26	8.78

Map 2, coverage 0.7, difference between the modes:

Mode	Mean run time (sec)	Path cost
M1	3.65	15.95
M2	100.31	27.37
M3	52.54	15.32