236901 - Algorithmic Robot Motion Planning

Sapir Tubul 305728180
Idan Lev Yehudi 206741878

February 2, 2022

2.2 - Robot Modeling

We will detail our approach for implementing each of the requested functions.

- `compute_distance` - We tested two different functions for computing distances between configurations. The first ("d_1") is simply the euclidean distance between the end effector in each of the configurations. The second ("d_2") is the sum of euclidean distances between each corresponding link in the two configurations. We tested paths generated with both methods, and ultimately chose d_2 because its paths were smoother. The intuition is that in d_1, two very different configurations can result in a 0 distance, if simply the end effector position is the same. On the other hand, d_2 will output 0 only if the configurations are exactly the same, and so close configurations according to d_2 are only those that require little movement in the workspace.
  In both of the cases, the workspace positions of each of the links are determined using the function `compute_forward_kinematics`, described later. The distance/sum of distances is computed using standard numpy functions.

- `compute_forward_kinematics` - Assuming the configuration given is (c_1, c_2, c_3, c_4), the links positions are computed iteratively according to the following formula:

$$\begin{pmatrix} x_i \\ y_i \\ \theta_i \end{pmatrix} = \begin{pmatrix} x_{i-1} + r_i \cdot \cos(\theta_{i-1} + c_i) \\ y_{i-1} + r_i \cdot \sin(\theta_{i-1} + c_i) \\ \text{wrap}_{[-\pi,\pi)}(\theta_{i-1} + c_i) \end{pmatrix}$$

  In words: each link position is the vector addition of the previous link position, and the current link length in the current direction. We assume that (x_0, y_0, θ_0) is the base of the manipulator.

- `validate_robot` - To validate that there are no self intersections of the robot, we construct a `Shapely LineString` object from the links position (computed using `compute_forward_kinematics`). A `LineString` object has an attribute is_simple, which is True when the `LineString` does not self intersect, so we simply return this value.

2.3 - Motion Planning

A few notes regarding the implementation of the following functions.

- `extend` - In $E1$ mode, we simply extend all the way to the sampled configuration. In $E2$ however, computing the extended configuration requires special care because the configurations are lists of angles, and angles are circular units. The extension algorithm is the same as when the configuration space is completely euclidean (i.e. simple vectors), with the addition of wrapping the angles to the range $[-\pi, \pi)$ each time vector subtraction or addition is computed between configurations.

- `compute_cost` - Simply the sum of distances between consecutive configurations in the path, according to the distance function `compute_distance`.

- `plan` - Implemented exactly like regular RRT, with the exception that the termination condition is based on having the end effector close enough to the requested location, and not based on distance of the entire configuration.

Goal biasing was implemented naively - simply sample the goal configuration with probability $p \in (0, 1)$. We test ran the algorithm with goal biasing of 0.05 and 0.2. We created a static visualization of the tree built, showing on the map the end effector positions only in the tree. The dynamic visualizations (gif) will be attached in the code.

Goal Biasing $p = 0.05$

Example trees visualized in figures 1, 2, and 3. Plotted in figures 4 and 5 are the graphs of success rate vs. compute time, and path cost vs. compute time. We can see that all of the plan times are in the order of magnitude of tenths of a second, and the path cost does not seem to be correlated to the computation time.

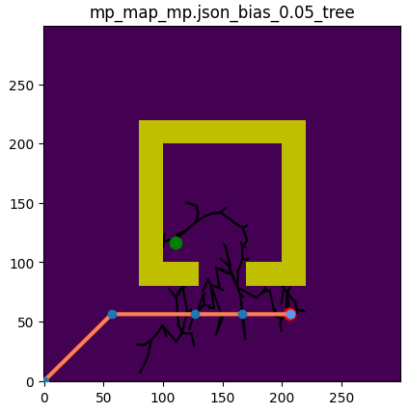

Figure 1: Example 1 for tree built visualization for motion planning, goal bias $p = 0.05$.

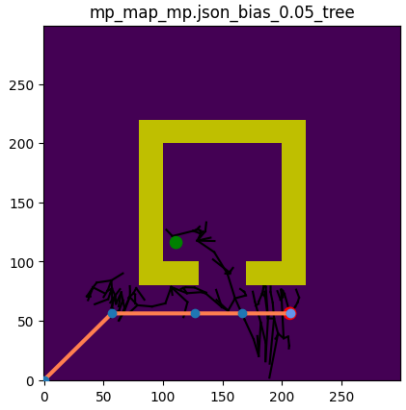

Figure 2: Example 2 for tree built visualization for motion planning, goal bias $p = 0.05$.

Goal Biasing $p = 0.2$

Example trees visualized in figures 6, 7 and 8. Plotted in figures 9 and 10 are the graphs of success rate vs. compute time, and path cost vs. compute time. We can see here that we had one outlier which took a significantly longer time to compute, and resulted in a longer path as well. In general, the higher goal bias produced worse results, with slightly higher average compute time and slightly longer paths. The results are summarized in the corresponding directories, in text files named in the convention $<$ `configuration` $>$ _ $<$ `goal_bias` $>$ `_summary.txt`.

2.4 Inspection Planning

In this section, we implemented an extension of RRT to account for points of interest seen along a path. Each vertex in the tree will hold a configuration x, and the set of inspection points seen along the path from the root to that vertex I_x. Like before, a random configuration x_{rand} is sampled, and the closest configuration to it in the current tree $x_{nearest}$ is found. We use the `extend` method to compute a new state

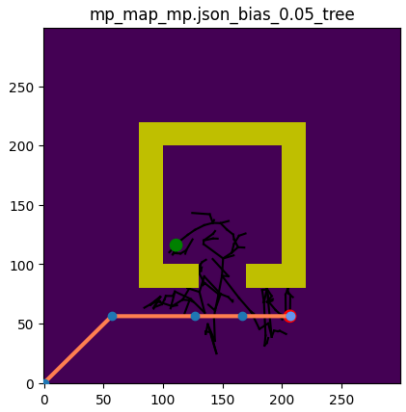

Figure 3: Example 3 for tree built visualization for motion planning, goal bias $p = 0.05$.

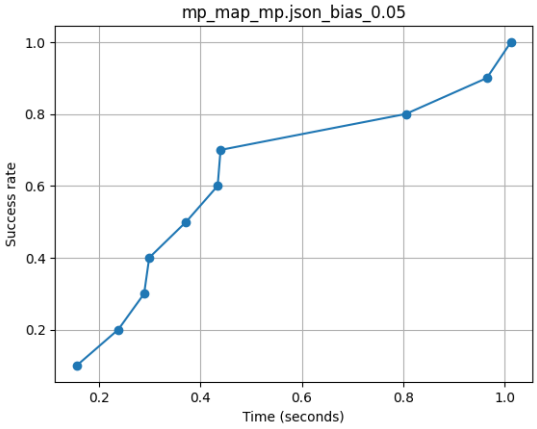

Figure 4: RRT motion planner success rate vs. compute time, goal bias $p = 0.05$.

x_{new}. If the configuration is connected to the tree, we augment x_{new} with the inspection points seen at that configuration: $(x_{new}, I_{x_{nearest}} \cup I_{x_{new}})$, and we add an edge in the tree connecting $(x_{nearest}, I_{x_{nearest}})$ with $(x_{new}, I_{x_{nearest}} \cup I_{x_{new}})$.

For goal biasing, we had to devise a method to generate configurations that view a certain target point. We constructed the following cost function, for given end effector pose and target point:

$$J\left(\begin{pmatrix} x_{ee} \\ y_{ee} \\ \theta_{ee} \end{pmatrix}, \begin{pmatrix} x_t \\ y_t \end{pmatrix}\right) = \left(\frac{1}{r_{max}}\sqrt{(x_t - x_{ee})^2 + (y_t - y_{ee})^2}\right)^2 + \left(\frac{2}{FOV}\left(\left|\underset{[-\pi,\pi)}{wrap}\left(\arctan 2\left(y_t - y_{ee}, x_t - x_{ee}\right) - \theta_{ee}\right)\right|\right)\right)^2$$

Having $J < 1$ implies that the end effector can see the target point. This is because $J < 1$ implies both of its summands are less than one (because they are strictly positive). Therefore

$$\left(\frac{1}{r_{max}}\sqrt{(x_t - x_{ee})^2 + (y_t - y_{ee})^2}\right)^2 < 1 \iff \sqrt{(x_t - x_{ee})^2 + (y_t - y_{ee})^2} < r_{max}$$

$$\left(\frac{2}{FOV}\left|\underset{[-\pi,\pi)}{wrap}\left(\arctan 2\left(y_t - y_{ee}, x_t - x_{ee}\right) - \theta_{ee}\right)\right|\right)^2 < 1 \iff \left|\underset{[-\pi,\pi)}{wrap}\left(\arctan 2\left(y_t - y_{ee}, x_t - x_{ee}\right) - \theta_{ee}\right)\right| < \frac{FOV}{2}$$

In the case $J < 1$, the distance between the end effector and the target point is less than r_{max}, and the angle between the end effector's optical axis and the direction to the target is less than $\frac{FOV}{2}$, meaning that the end effector views the target point.

We then implemented a gradient descent to minimize this cost function until a value of less than 1 is reached.

To sample a goal configuration, we randomly choose a target inspection point. We then use the current best configuration as a starting configuration, and gradient descent until $J < 1$, a maximum number of iterations is reached, or the cost function is increasing. From our debugging tests, in about 60% of the time the gradient

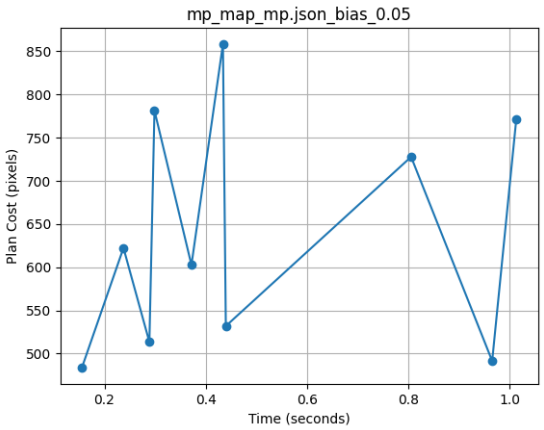

Figure 5: RRT motion planner path cost vs. compute time, goal bias $p = 0.05$.

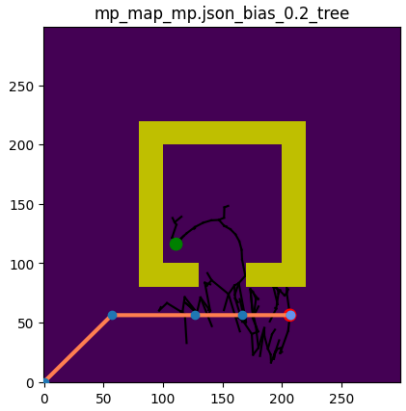

Figure 6: Example 1 for tree built visualization for motion planning, goal bias $p = 0.2$.

method succeeds in producing configurations which view the target point. We assume in most part the remaining 40% fails due to obstacles. We observed that only a small part fails due to convergence failure, even though we implemented a naive gradient descent.

We test ran the algorithm with target coverage requirements $c = 0.5$ and $c = 0.75$.

Coverage $c = 0.5$

Example trees visualized in figures 11, 12, and 13. Plotted in figures 14 and 15 are the graphs of success rate vs. compute time, and path cost vs. compute time.

Coverage $c = 0.75$

Example trees visualized in figures 16, 17, and 18. Plotted in figures 19 and 20 are the graphs of success rate vs. compute time, and path cost vs. compute time. Note: in the first run we had the runner getting stuck for several hours. We added the condition that if the planning time exceeds one hour, then the tree built is reset and search continues from scratch. This helped escape from bad tree settings, and as evident in the summary graphs, there was one run which took slightly longer than one hour.

Overall, the RRT algorithm is not suitable as-is for inspection planning. Even in this simple example where all inspection points are close to each other, searches get stuck (computation time exceeds 10 mins) quite often. This is because the RRT algorithm is designed for producing linear paths from start to goal, however in inspection planning we often need to find a path which revisits vertices as it attempts to cover all inspection points. As we saw in class, the voronoi bias of the RRT implies that the probability of extending the leaves of the tree towards the unexplored regions of the C-space is much higher than extending them towards already explored regions. This means that the probability the existing optimal plan in the tree will be extended back to an already explored region (as required by inspection planning) is practically non existent.

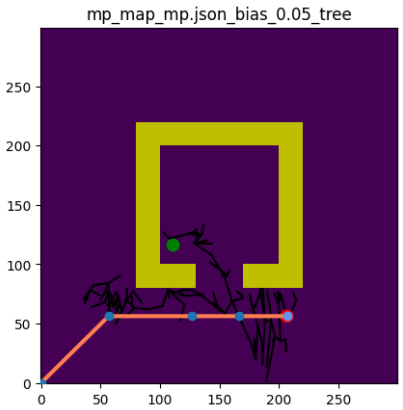

Figure 7: Example 2 for tree built visualization for motion planning, goal bias $p = 0.2$.

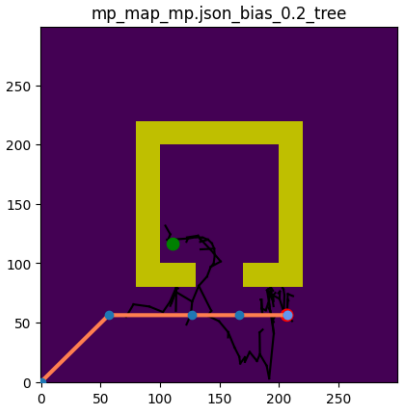

Figure 8: Example 3 for tree built visualization for motion planning, goal bias $p = 0.2$.

2.5 Competition

We wish to participate in the competition.

For the competition, we tested several improvements to the RRT inspection planner algorithm described above.

The first was adding a rewire operation, similar to RRT*. Since we observed that the tree building often gets stuck (especially in the high desired coverage setting), we wanted to make connections to the best already available path more probable, to reduce the probability of having two contesting paths which prevent each other from growing into the region occupied by the other. We tested several flavors of the rewire, differing in the condition based on which to connect $x_{potential\_parent}$ and x_{child}. In all of the cases, we chose the number of nearest neighbors to check for rewire to be logarithmic in the size of the tree.

1. Rewire if both the candidate path from $x_{potential\_parent}$ to x_{child} has length that is strictly shorter, and its coverage is equal or greater. This is the quite conservative, however forgoing one of the conditions does not work. Not checking for path cost improvements results in having cycles in the tree, and not checking for coverage results in no paths found because the rewire hurts the coverage, and ultimately makes the planner fail to find a path. This method results in a longer average computation time but lower rates of the planner getting stuck, and the paths generated are shorter on average.

2. Rewire if (1) holds, or if $x_{child}$is a leaf and the candidate path has a strictly greater coverage. This resulted in very long paths (about x2 longer), but shorter computation times. The paths tended to be very jaggy around inspection points, as the planner preferred improving coverage over cost when adding new states.

In the end, the rewire operation seemed to improve the path quality, but did not improve significantly enough the chances of the planner to get stuck. It caused a major increase in the computation time, so we decided to remove it.

The second improvement we tested was a dynamic goal bias. We postulated that when the planner is successful in extending new states, it should be more biased in moving towards the goal, and when it fails to extend new states, it should randomly search for new states to help it get out of the local minima it is stuck

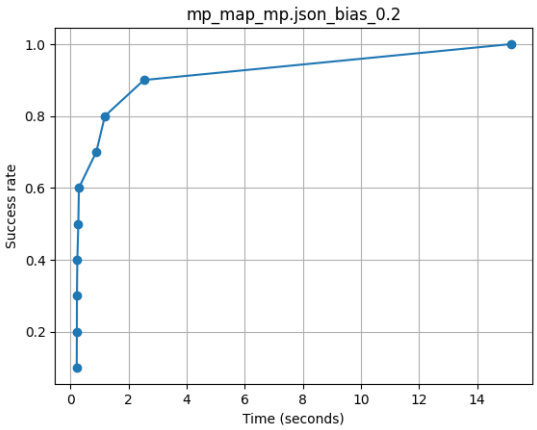

Figure 9: RRT motion planner success rate vs. compute time, goal bias $p = 0.2$.

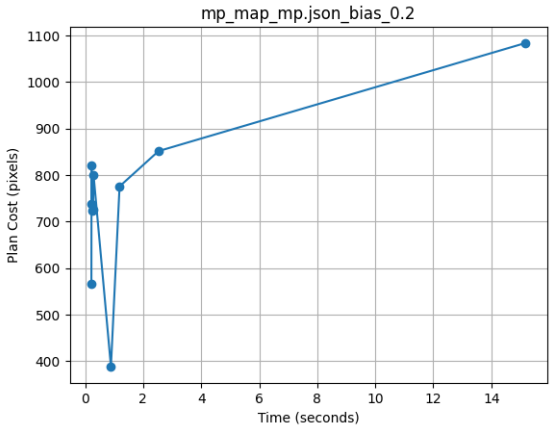

Figure 10: RRT motion planner path cost vs. compute time, goal bias $p = 0.2$.

in. We compute the extension success rate from $x_{nearest}$ to x_{new} over a sliding window (tested several window sizes between 50 and 1000, currently chosen 100). We then set the goal bias as the computed extension success rate, times a weight constant to ensure that we always have random sampling. We tested several weight factors and chose 1.

In the goal biasing, we tried several heuristics for choosing a target inspection point: randomly chosen from the points not seen along the path to the current best configuration in the tree, or deterministically choosing among them the closest one to the best configuration. It seemed like deterministic choices were worsening the running time, and also increased the chances of getting stuck in the tree, so we went with the random choice.

The last thing that we added for the competition to circumvent the RRT not being able to find paths that revisit the same locations was inner and outer iterations. In inner iterations, we build the RRT in the regular scheme. Every k inner iterations, we check to see if we found a path that new inspection points. If we haven't, we continue for another k iterations. If we have, we return the path found so far. The outer iterations reset the tree, setting the start configuration to be the end configuration of the path returned, and excluding from the set of inspection points those covered by the path. The outer iterations continue until all inspection points have been covered, or until a maximum total number of iterations has been reached, to avoid special cases where there are some inspection points that cannot be reached.

This method worked suprisingly fast, at the expense of path quality (about x2 increase). However, in about 95% of the cases, the paths returned had 100% coverage.

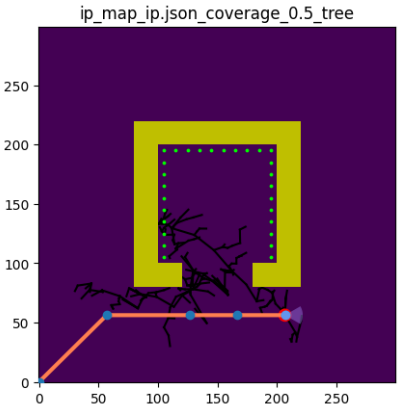

Figure 11: Example 1 for tree built visualization for inspection planning, coverage $c = 0.5$.

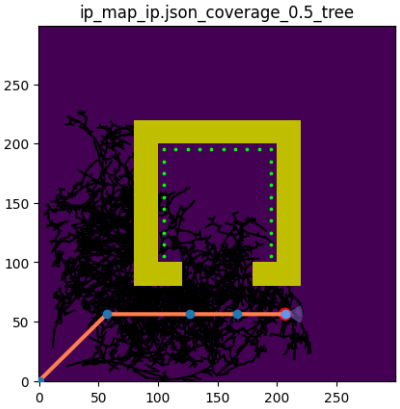

Figure 12: Example 2 for tree built visualization for inspection planning, coverage $c = 0.5$.

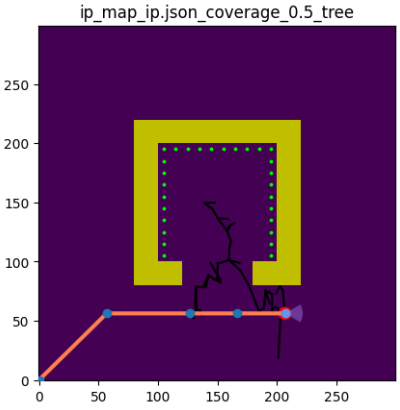

Figure 13: Example 3 for tree built visualization for inspection planning, coverage $c = 0.5$.

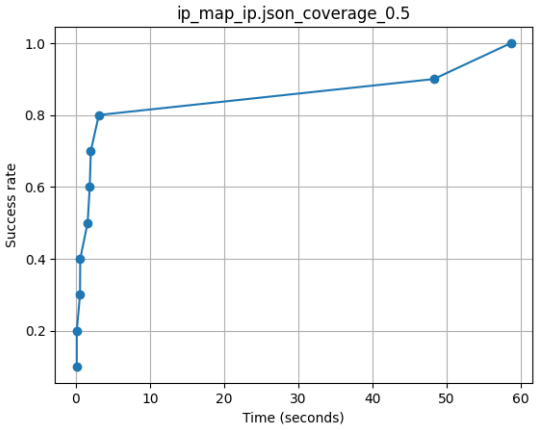

Figure 14: RRT inspection planner success rate vs. compute time, coverage $c = 0.5$.

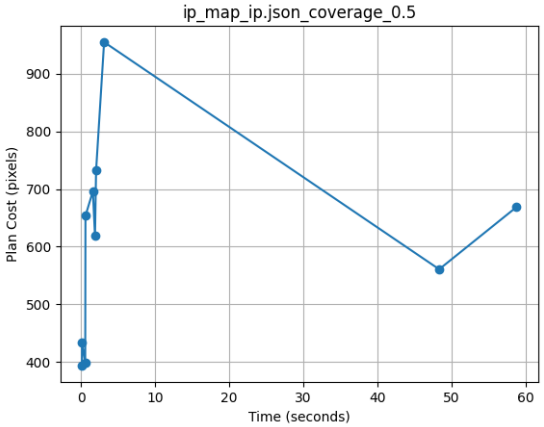

Figure 15: RRT inspection planner path cost vs. compute time, coverage $c = 0.5$.

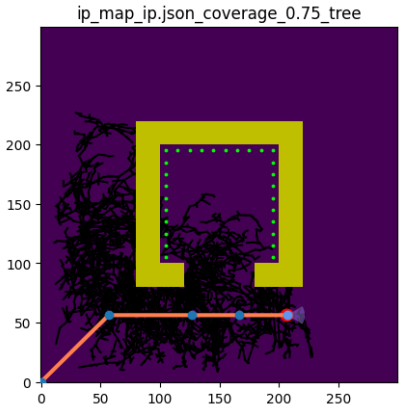

Figure 16: Example 1 for tree built visualization for inspection planning, coverage $c = 0.75$.

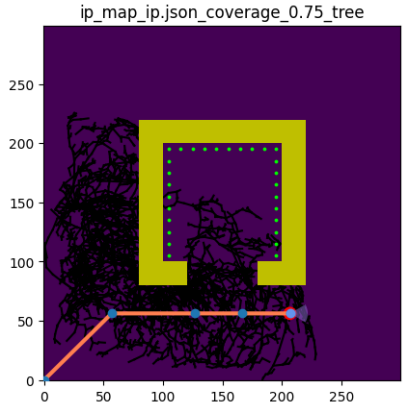

Figure 17: Example 2 for tree built visualization for inspection planning, coverage $c = 0.75$.

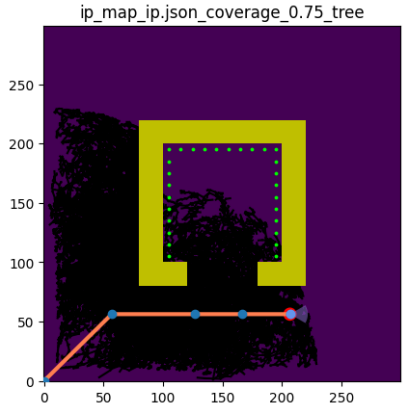

Figure 18: Example 3 for tree built visualization for inspection planning, coverage $c = 0.75$.

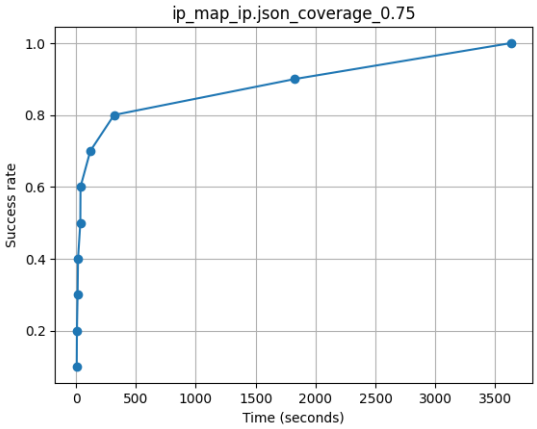

Figure 19: RRT inspection planner success rate vs. compute time, coverage $c = 0.75$.

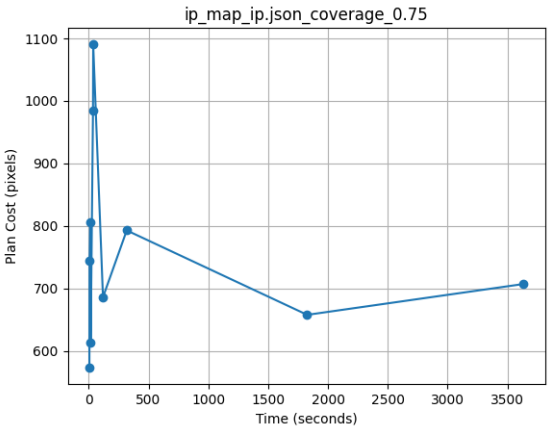

Figure 20: RRT inspection planner path cost vs. compute time, coverage $c = 0.75$.