hw5

Name: Ruffin White
ID: A53229660

March 23, 2018

Contents

1 HW5 : Math for Robotics **2**
1. Configuration Space 2
2. Greedy Search 6
3. Safest Path 9
4. Probabilistic Road Maps 10
5. Rapid Exploring Random Trees 14
Summery 15

HW5 : Math for Robotics

Author: Ruffin White
Course: CSE291
Date: Mar 23 2018

The world model is shown in figure 1. The robot is a differential drive system of size 50x50.

1. Generate the configuration space for the robot with a grid size of 2x2 and 5 deg in angular resolution. Generate an illustration of what the configuration space looks like with the robot at orientations 0, 45 and 90 deg.
2. Use greedy search find the shortest path between start-point (50,50) and end-point (750,250). Illustrate the path and provide its length.
3. Compute the safest path from start to finish (hint: medial axis transform). Illustrate the path and provide its length.
4. Use probabilistic roadmaps (PRM) to compute a path between startand end-points with 50, 100 and 500 sample points. What is the difference in path length? Illustrate each computed path.
5. Do the same with Rapid exploring random trees (RRT). What are the main differences in performance between PRM and RRT? Illustrate each path.

1. Configuration Space

We'll start by defining a grid-based environment to encapsulate a discretized representation of our world model. We then will populate the 2D grid environment with the set number of wall objects that constrain possible paths what limits our work space.

Well also define our Robot as pixelated shape, subject to the same scale used in deriving the environment.

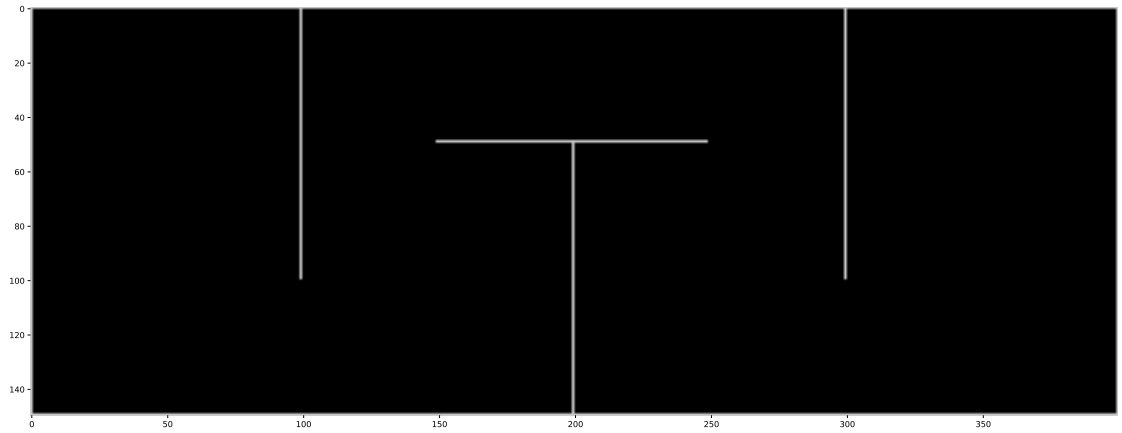
Here we'll define a function to rotate our robot representation given an angle in degrees.

Given both are world and robot representations are represented as occupancy and 2D arrays, we can use simple 2d convolution, optimized with the FFT transform, to generate our configuration space with respect to a given angle, or list of angles In which the robot is rotated.

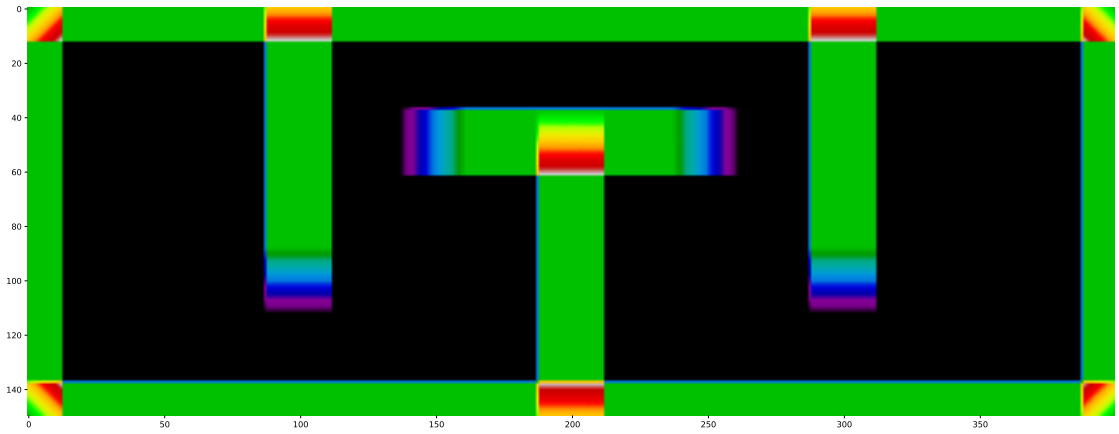
We also define a simple function to plot our workspace or partial configuration space for visualization.

Here will generate our configuration space for the angles between 0 and 90 degrees at a degree resolution of one. Because we are using scipy's fftconvolve function that padds our signal to next nearest power of the hamming numbers for the fft function optimized for those radixs. This makes this step quite faster than compared to traditional 2D convolution of signal/kernel or workspace/robot respectively.

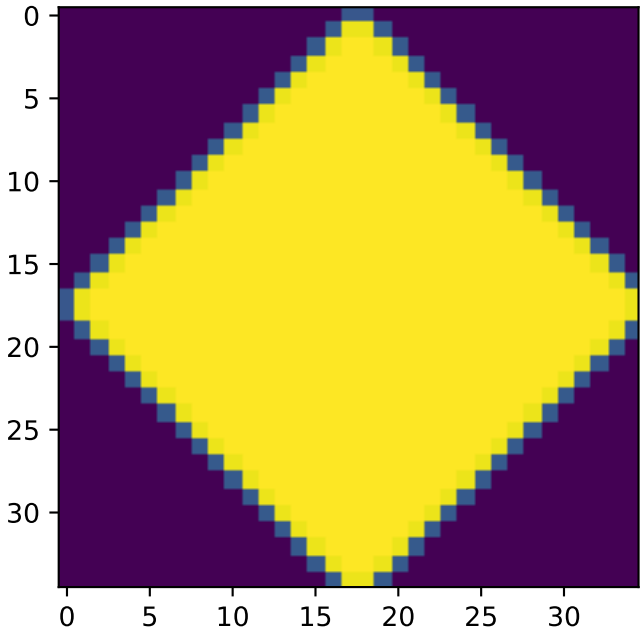
Plotting this grid based world, we can see the layout as shown in the assignment figure.

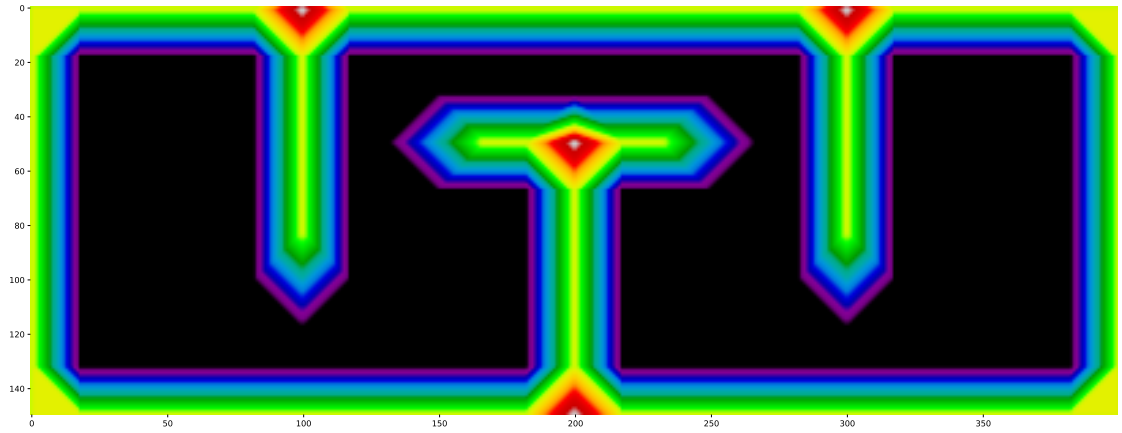


In the next three figures, we can visualize the configuration space 4 robot rotations at 0, 45, and 90 degrees respectively. Because our configuration space was acquired using to the convolutions, we can visualize the raw response to provide additional insight in robot collisional intensity, as well as a regionalized cost map.

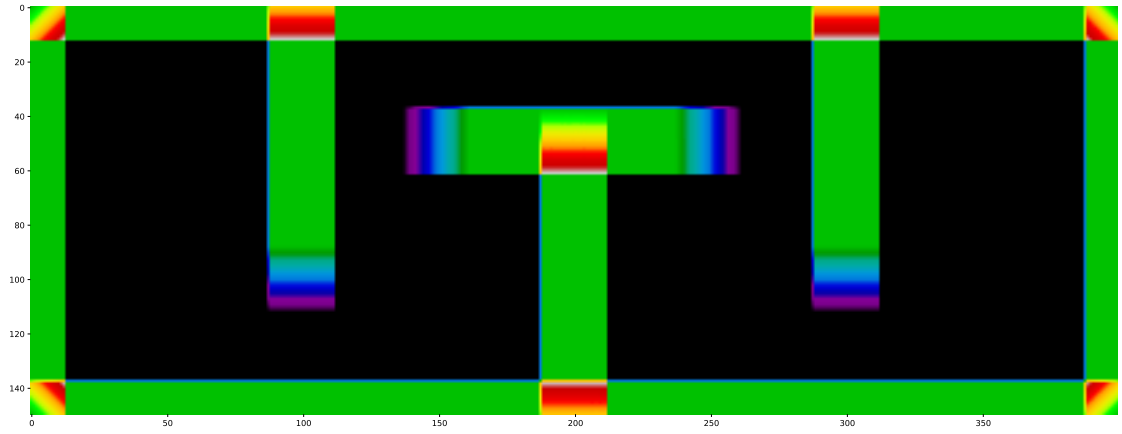


The kernel for the robot's footprint rotated at 45 degrees.



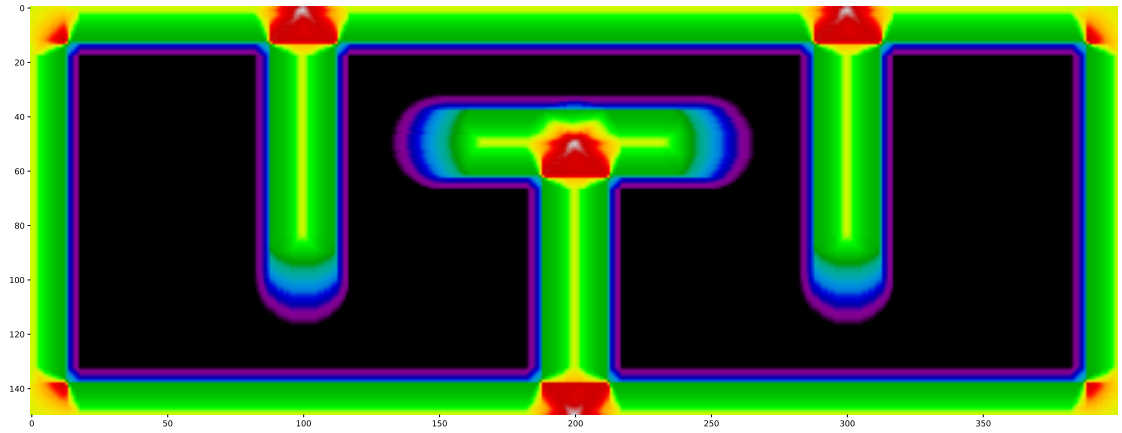


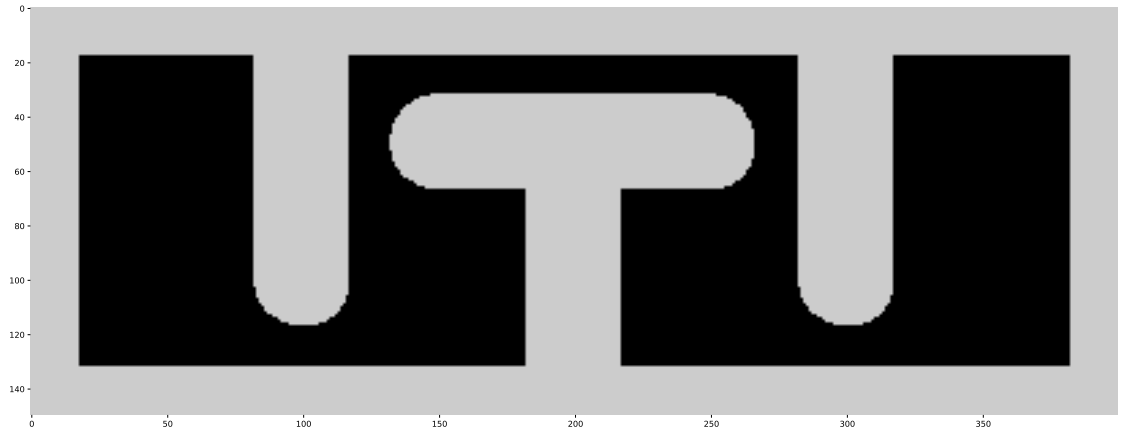
Above is the configuration space for the 45 degree rotation, while below is the space for 90 degree rotation and is visually similar to the 0 degree rotation.



Because our robot is rotationally symmetrical across every half pi radians, we can flatten the configuration space across rotations from 0 to 90 degrees by taking the max value along the 3rd axis on our configuration space 3D array representation. This renders the most conservative configuration space where the black region denote all points in the world reference frame that our robot can freely rotate in place.

This is useful for robotic navigation stacks that commonly make use of rotational recovery behaviors when losing localization or determining alternate routes. The astute reader will notice that this is the same result as in taking the the 2D deconvolution of the world map with respect to a kernel that is a simple Circle with a radius defined as the maximum distance a collision point along the robot geometry is with respect to the center of its rotation (assuming our differential drive model).





We can define a function to sweep through the configuration space in an animated fashion. However this is quite slow using to matplotlib.

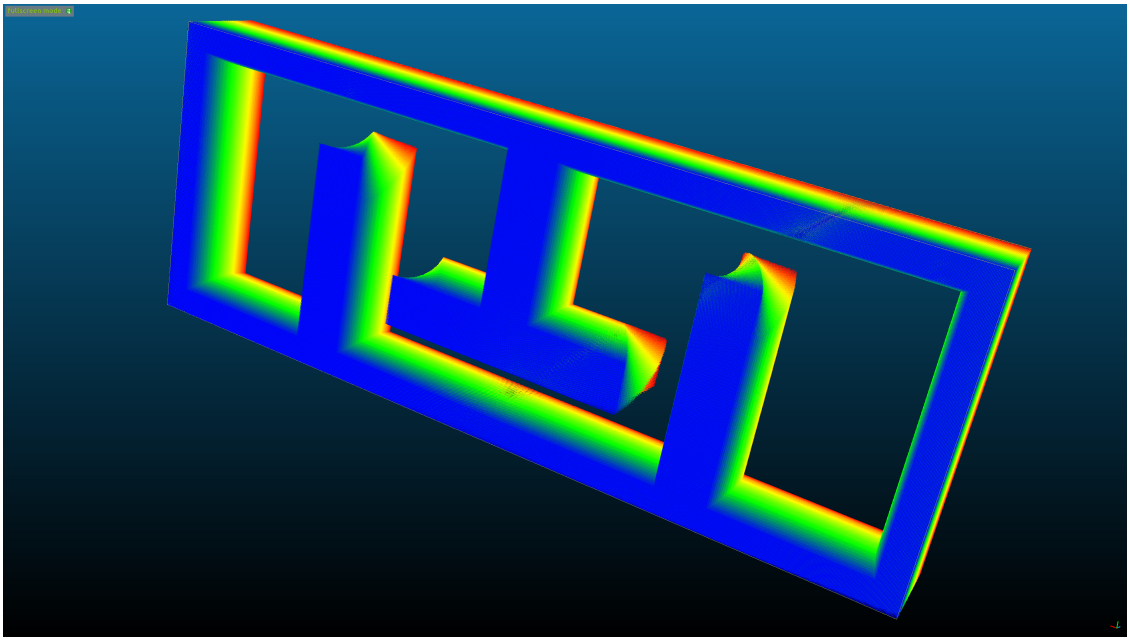
You can take a look at animation by viewing figs/config_space.mp4 that depicts the space by moving along the rotational axis for the multi-dimensional configuration space.

```
<IPython.core.display.Video object>
```

A better alternative is perhaps using pygtgraph and let OpenGL do the heavy lifting for us in rendering the frames.

We can also use pygtgraph to slice the Configuration space arbitrarily, similarly done with MRI scans.

Lastly we can export the occupancy grid from our configuration workspace and to a three dimensional Point cloud. Obviously this only works because of the 2D dimension of our initial world frame with the additional dimension of rotation. However the same slicing technique, as above, could be used to introspect the configuration space for higher dimensions.



2. Greedy Search

For simple search, we can use basic methods such as A, *or due to the grid-based regularity of the search domain, we can leverage more advanced methods such as Jump Point Search (JPS). JPS take advantage of the grid based symmetry by recursively traversing in the direction of expansion until a stop condition is reached. Stop conditions for JPS mainly pertain to instances where symmetry along either side of the expansion direction is broken. This stop point is then used as a seed for the expansion point. JPS can also leverage heuristics same as A*, where our heuristic here is made admissible by utilizing the euclidean distance between the expansion point and goal.

Here will simply export our configuration space into a yaml file suitable for a C++ JPS implementation to search through. We then can read back the solved paths and compare A* to JPS. The source code for this executable can be found here at this fork

An attempt was made to search the full configuration space using JPS in 3D, as exampled here, however because we normally want to optimize for distance traveled within the workspace as opposed to the configuration space, the cost for traversing the configuration space must also reflect this flexibility. Doing this would require a zero edge cost and traversercing the configuration space through the rotational dimension, permitting the robot to freely rotate on its axis without penalty. Additionally, because the robot here is not holonomic, we can't necessarily move in every direction at any given point in the configuration space because of the restriction of the robot dynamics. This would necessitate an alternate formulation of the discrete configuration space representation that would not correspond to a regular lattice like graph.

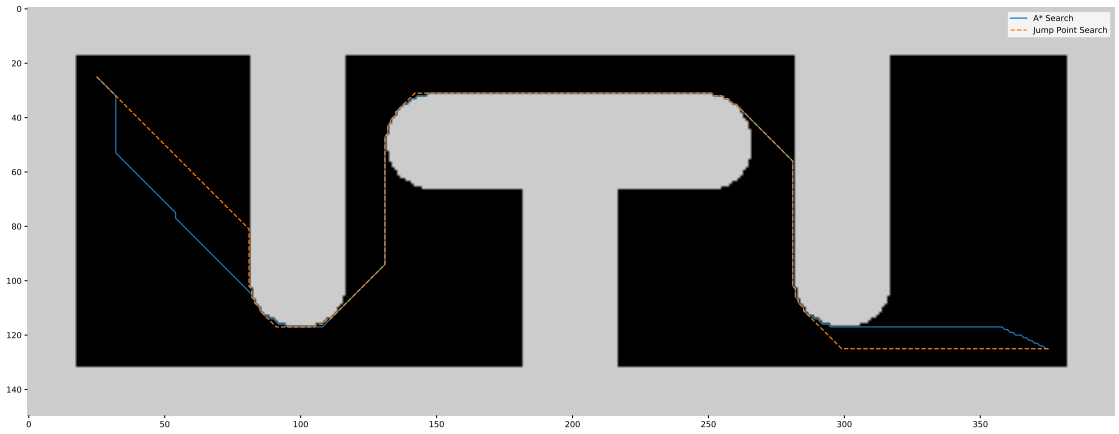
Thus, for simplicity and runtime complexity using python functions, the configuration space has been compressed or projected into a 2D subspace for quick planning. In practice this could potentially remove all potential solutions, however in this case we can visually verify that even the most conservative representation is still adequately solvable.

```
start: 25 25
goal: 125 375
origin: 0 0
dim: 150 400
resolution: 1
JPS PLANNER VERBOSE ONStart: 25 25
```

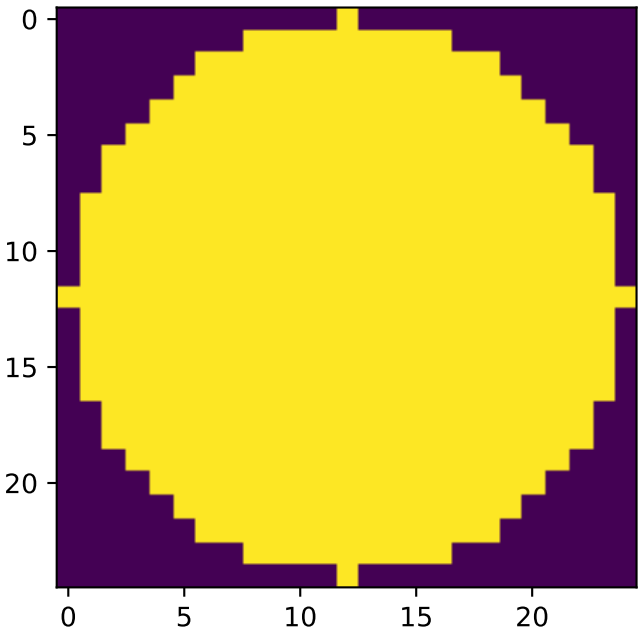
```
Goal:  125 375
Epsilon:  1
Goal Reached!!!!!!

goal g: 538.232539, h: 0.000000!
Expand [841] nodes!
JPS Planner takes: 0.000000 ms
JPS Path Distance: 538.232539
AStar Planner takes: 12.000000 ms
AStar Path Distance: 538.232539
```

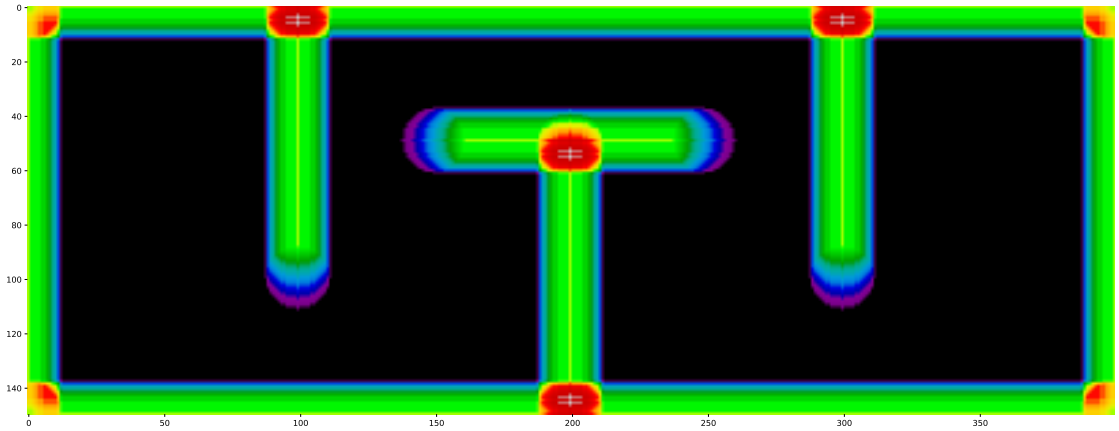
From the details printed above, we can see the distances of the two paths found using A* and JPS are equivalent in length, however JPS took significantly less time to find the optimal grid based path for this particular subspace.

However, because of the Triangle inequality, we know the paths generated from our Manhattan based configuration space will always be of a longer distance than that of pure continuous euclidean solutions. The distance here however can provide a suitable upper bound for any euclidean based methods such as those using high-resolution visibility graphs.



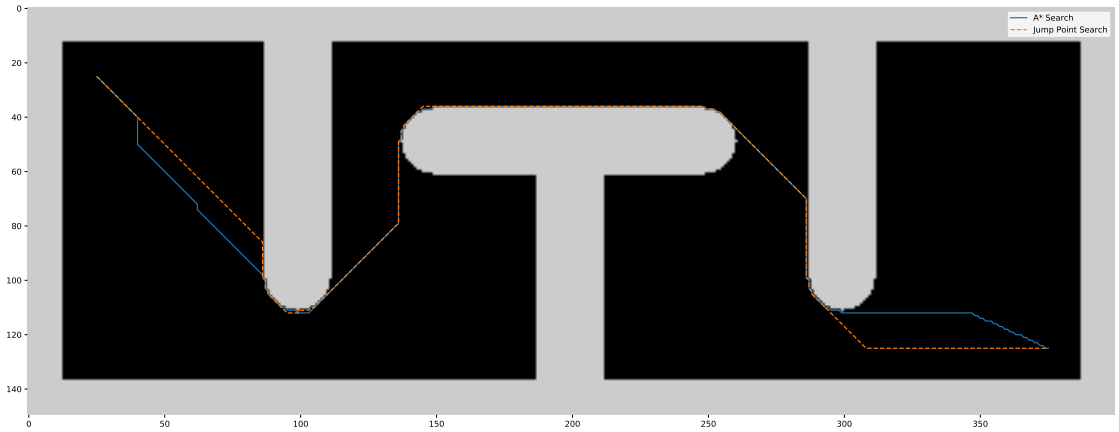
To tighten this upper bound of optimal path distance even further, we can instead consider the more optimistic projection of the configuration subspace by convolving with only the inner diameter of the robot. The intuition here being: as would a race car driver hug the inner corner of any turn to minimize distance travel along a route, so too should the differential robot hug the inner wall of obstacles when circumnavigating corners. However, because our traversal of the discrete configuration space is limited to bearings along long quarter pie increments, we know this solution to still be suboptimal. A geometric path planner using euclidean space would most likely return a path with ray traced trajectories that tangentially intersect the turning points show here.





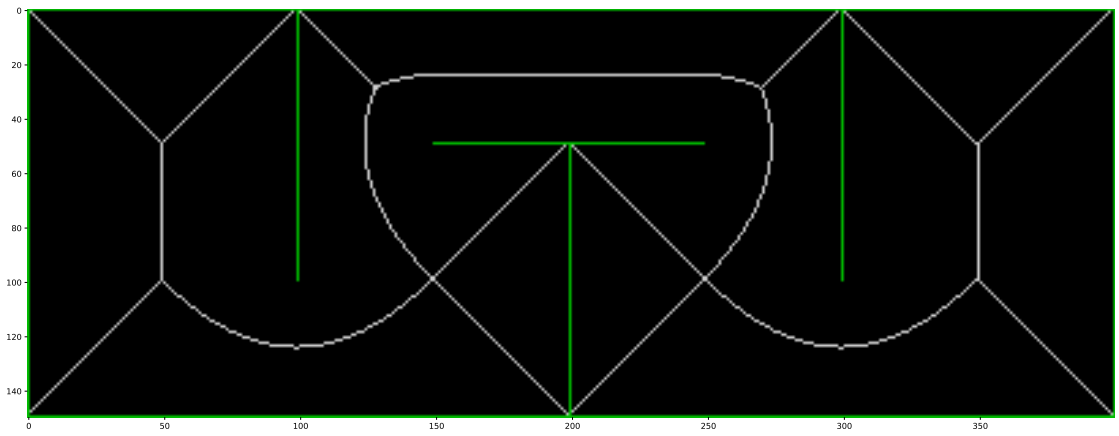

```
start: 25 25
goal: 125 375
origin: 0 0
dim: 150 400
resolution: 1
JPS PLANNER VERBOSE ONStart: 25 25
Goal:  125 375
Epsilon:  1
Goal Reached!!!!!!

goal g: 503.587878, h: 0.000000!
Expand [718] nodes!
```

```
JPS Planner takes: 1.000000 ms
JPS Path Distance: 503.587878
AStar Planner takes: 19.000000 ms
AStar Path Distance: 503.587878
```



3. Safest Path

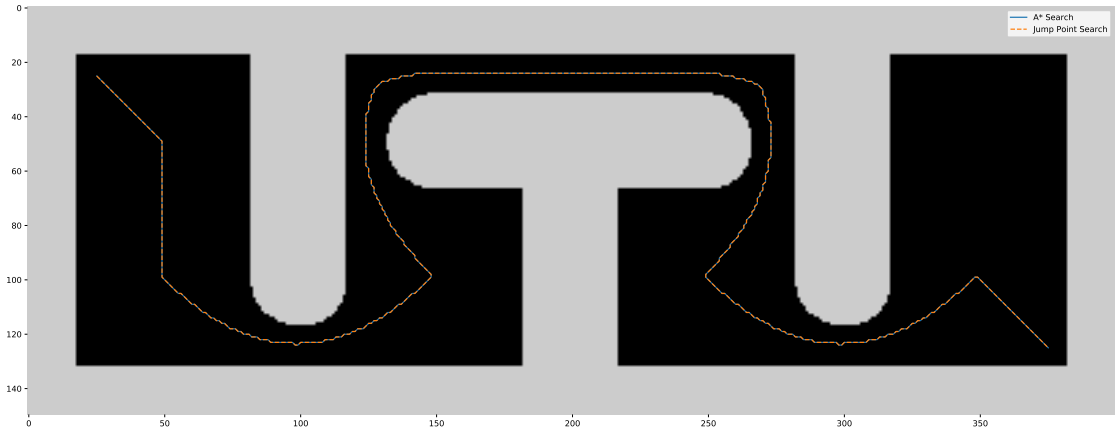
Computing the safest is path through our workspace environment will in this case equate as being the trajectory that is maximally distant from all nearby obstacles. Essentially this necessitates computing the voronoi partitions for the empty regions in the work space. We can then travers the set of points generated from this tessellation as it provides us a simplistic roadmap. We then need only find the closest decomposed points to both the start and the goal. As a final check, we can sample the resulting trajectory using our configuration space as a collisions look up.



```
start: 25 25
goal: 125 375
origin: 0 0
dim: 150 400
```

```
resolution: 1
JPS PLANNER VERBOSE ONStart: 25 25
Goal:  125 375
Epsilon:  1
Goal Reached!!!!!!

goal g: 669.470129, h: 0.000000!
Expand [879] nodes!
JPS Planner takes: 0.000000 ms
JPS Path Distance: 669.470129
AStar Planner takes: 0.000000 ms
AStar Path Distance: 669.470129
```

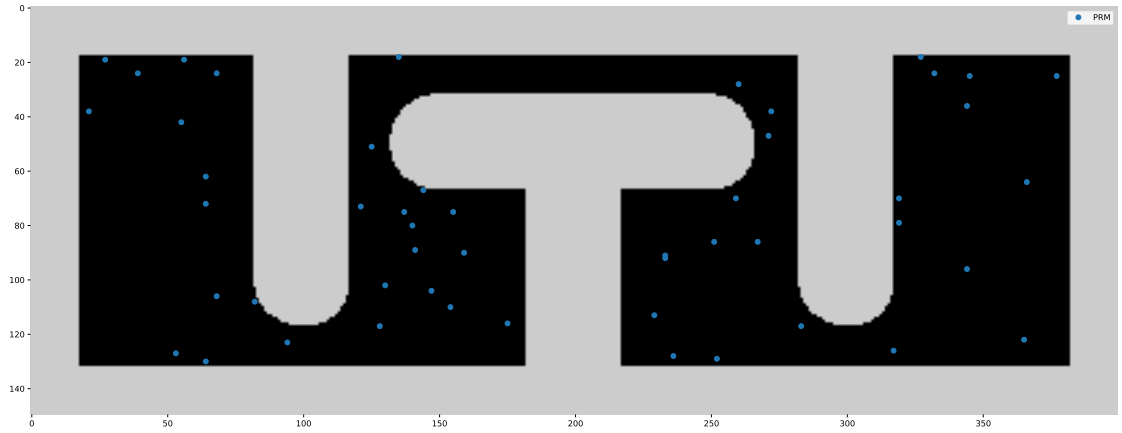
From the results shown we can see that the trajectory following the voronoi partitioning is well far away from any congestion points with some margin of room to spare. This could be useful for any kind of trajectory controller that has a good amount of play and its set point pursuit, permitting some amount of deviance from the setpoint trajectory to enable higher speed dynamics.



4. Probabilistic Road Maps

Position for meditation and probabilistic roadmaps, will simply sample the unoccupied free space in the discrete projection of the configuration subspace, Then attempt to connect a road map between unobstructed points and then finally search the graph for a suitable trajectory.

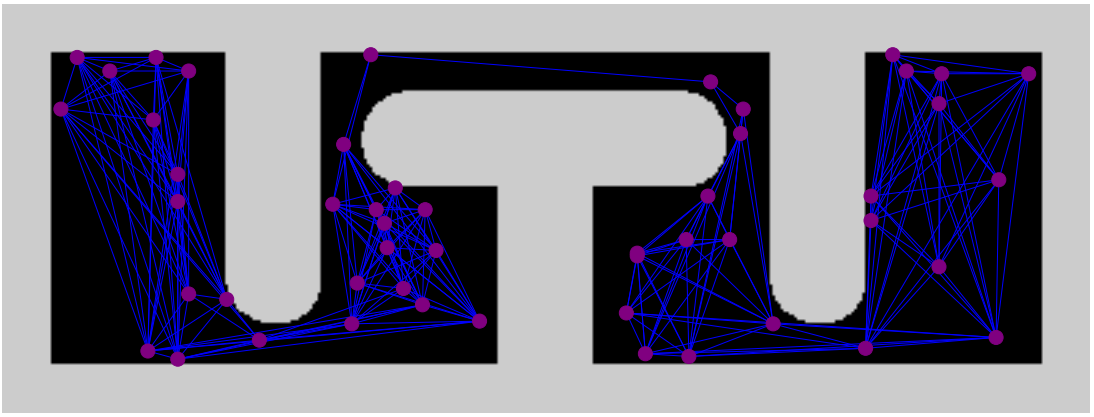
We can simplify the sampling method by flattening the projected subspace into a 1D array, and then sampling from that array using the occupancy values as probabilistic weights of the point to be sampled from.



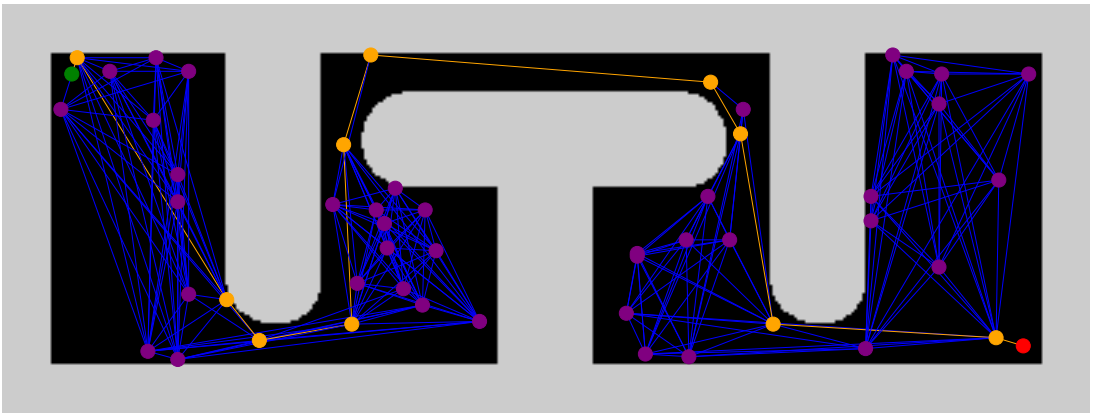
Now that we have a number of samples, will construct a fully connected graph from the samples, then traverse along their edges to ensure they are unobstructed, and if not simply prune them from the roadmap.

```
Number of edges when completely connected:  1225
Number of edges after pruning:  1225
```

Notice that the number of edges for a completely connected graph is $\frac{n\cdot(n-1)}{2}$, where in is the number nodes. For $n = 50$, we find the number of edges to be 1225. This Exponential of complexity will soon catch up with us as we increase the number of samples for a roadmap. Because collision checking is done on an edge by edge basis via boolean logic across profile lines cutting the Configuration Subspace, reducing the number of edges later will be important and minimizing this bottleneck to achieve higher-density road maps.



To use and search the road map, We simply find the respective nearest unobstructed PRM neighbors for the start and goal to attach them to the graph, Then simply use heuristic based search method such as A* to traverse the graph and return a shortest path.

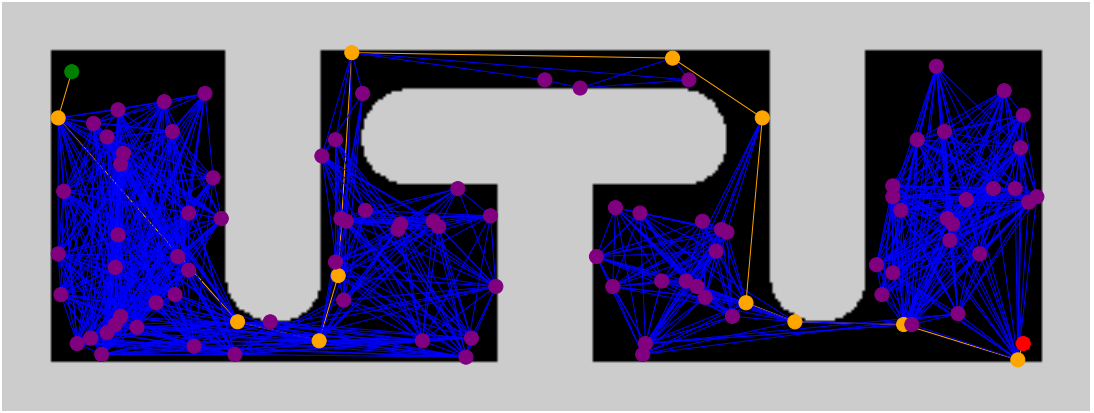


```
PRM Length:  576.2000283301505
```

When considering the probabilistic roadmap using twice as many samples, we know that the number of edges will roughly quadruple. To mitigate this we can probabilistically prune the complete graph by randomly removing edges, where the likelihood of removing an edge is directly proportional to its length. We also finesse this distribution weights a little to ensure that shorter edges have a lesser chance of removal.

The intuition here is that longer edges have a higher likelihood of intersecting obstacles, and are thus less likely to be unobstructed and can be preemptively culled from the politician without wasting time in checking. This has the potential drawback of increasing the likelihood that the roadmap graph is disjoint, however the complexity savings easily afford us the means to increase the sampling density to compensate. Here we'll use a dropout of 50% to speed up the quiry.

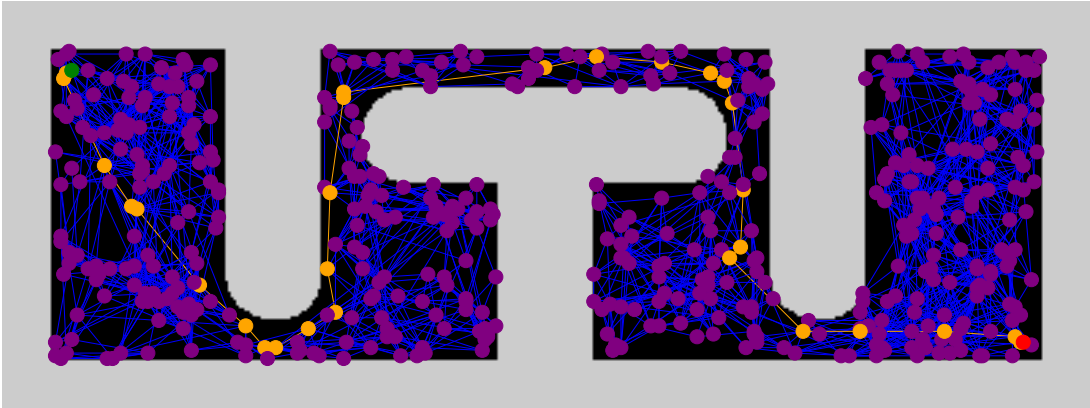
```
Number of edges when completely connected:  4950
Number of edges after pruning:  2475
```



```
PRM Length:  591.1415029610863
```

For the case with 500 samples, we use a more aggressive pruning ratio of 98%.

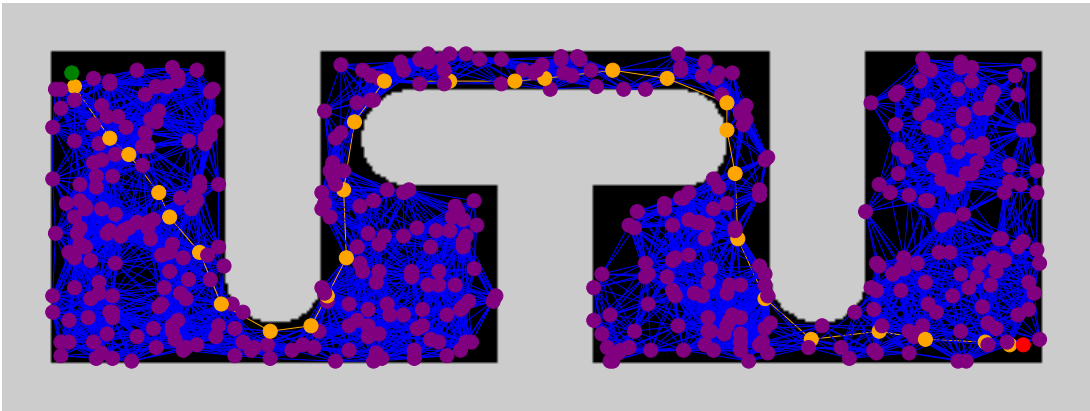
```
Number of edges when completely connected:  124750
Number of edges after pruning:  2495
```



```
PRM Length:  570.698263078384
```

An alternate approach to pruning probabilistically is to simple remove the n longest edges using a heap sort method. This has the added benefit that the only paths remaining are minimally short, and are thus also quick to compute collision checks. The geometric regularity of this approach also ensures that nearby points are more robustly connected, preventing unnecessary detours and circumnavigating nodes of close proximity. The downside however is that you are more likely to remove longer edges that could provide single shot shortcuts, or again, potentially bisect the road map graph; where the distances between the cuts of two subgraphs is relatively large In comparison to their inner edge distances.

```
Number of edges when completely connected:  124750
Number of edges after pruning:  6238
```



```
PRM Length:  539.3596836098676
```

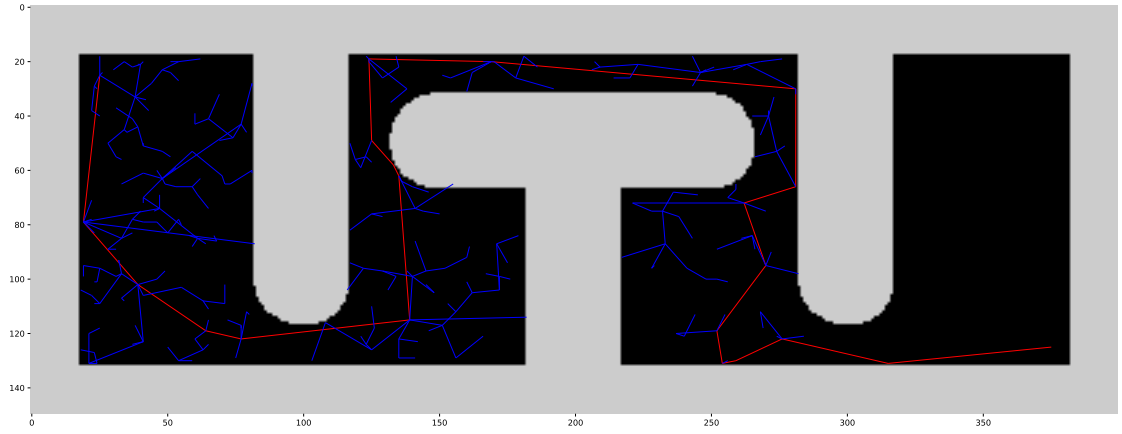
5. Rapid Exploring Random Trees

For our RRT, we can again use the configuration subspace has a quick collision look up table while we continue to sample free space points on the fly. When generating the tree from the sample points, will use a variant that is goal connect directed. This means that the pursuit of some sample will not be terminated prematurely due to some finite delta, but will instead extend as far as possible until the sample is reached for an obstacle is encountered.

This method here is also goal-directed given that with some probability the goal is selected as a candidate sample for extension, preferably at some larger degree then random subspace. The probabilistic bias towards the goal is itself a trade-off question, where biasing towards the goal to much can results and fruitless extensions into obstructing concave obstacles, or to little resulting in fruitless exploration of the entire subspace even after approaching the goal.

```
2  Bumbed wall:  [82 87]
103  Bumbed wall:  [182 114]
146  Bumbed wall:  [133  58]
149  Bumbed wall:  [155  65]
278  Bumbed wall:  [281  30]
295  Bumbed wall:  [116 104]
323  Bumbed wall:  [282  98]
476  Bumbed wall:  [266  55]
Goal found!
Iterations taken:  568
```

```
Out[78]: True
```



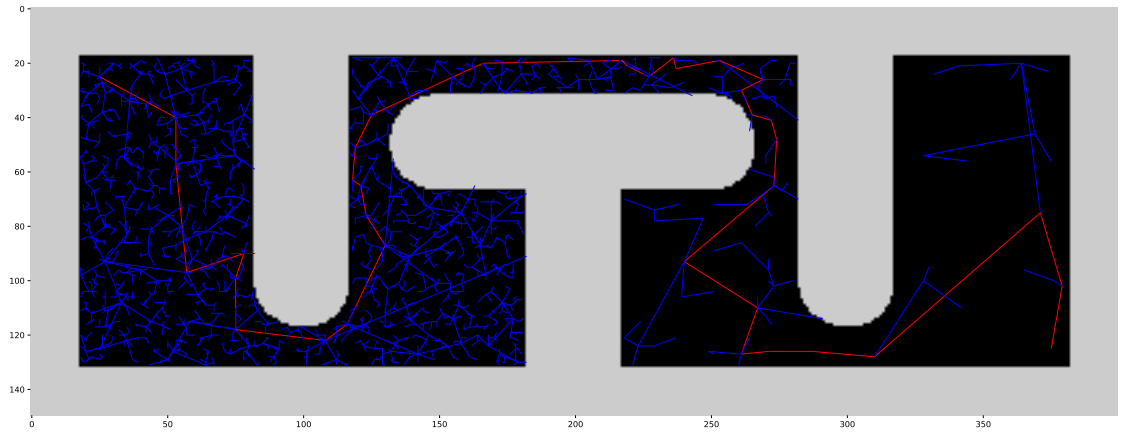

```
RRT Length:  694.5711277181326
```

The RRT method above also re-balances the sample space after each sample. because we're working with a discretize space, we can avoid double sampling By waiting be extended coordinates as such to remove them from the sample pool. this speeds of computation a little and also prevents the tree from becoming unnecessarily dense near the starting point of the tree. Below is an example with re-balancing disabled, note the dense foliage near the root of the tree.

```
4  Bumbed wall:  [82 59]
13  Bumbed wall:  [82 90]
```

```
63  Bumbed wall:  [182  91]
118  Bumbed wall:  [133  55]
167  Bumbed wall:  [163  65]
290  Bumbed wall:  [182 130]
402  Bumbed wall:  [182  68]
763  Bumbed wall:  [243  32]
2001  Bumbed wall:  [282  41]
2460  Bumbed wall:  [264  45]
2882  Bumbed wall:  [281  20]
3044  Bumbed wall:  [257  33]
3916  Bumbed wall:  [282  70]
3970  Bumbed wall:  [291 114]
Goal found!
Iterations taken:  4239
```

```
Out[80]: True
```




```
RRT Length:  750.6010632020418
```

Summery

From the summarizing table below, we find that greedy based search methods for discretized configuration spaces provide a tighter upper bound on optimal trajectories and can computed quite fast when leverage the symmetrical structure of the search space. Voronoi Partitioning can also be computed quite fast and can be solved for even quicker due to its narrowed scope in reachability, however results and longer found paths due to its conservative nature.

PRMs offer another fast method when correctly tuned with appropriate pruning parameters given that the configuration space is known beforehand and can be sufficiently sampled, where path optimality is often proportional to sample density/ coverage. However, if the configuration space is not known beforehand, a more exploratory method such as RRTs provide useful search method for expanding out from a known start point and into an uncertain environment. The search domain here, given the configuration space was discreet and completely known, does not necessarily play to RRT strengths where normally the configuration space would only need to be sparsely computed during online exploration, but can still be made applicable with offline lookups.

```
<IPython.core.display.HTML object>
```

Method	Distance
A*/JPS conservative	538.233
A*/JPS optimistic	503.588
voronoi partitions	669.47
length_prm_50	576.2
length_prm_100_prob_prune	591.142
length_prm_500_prob_prune	570.698
length_prm_500_sorted_prune	539.36
length_rrt_rebalance	694.571
length_rrt_no_rebalance	750.601