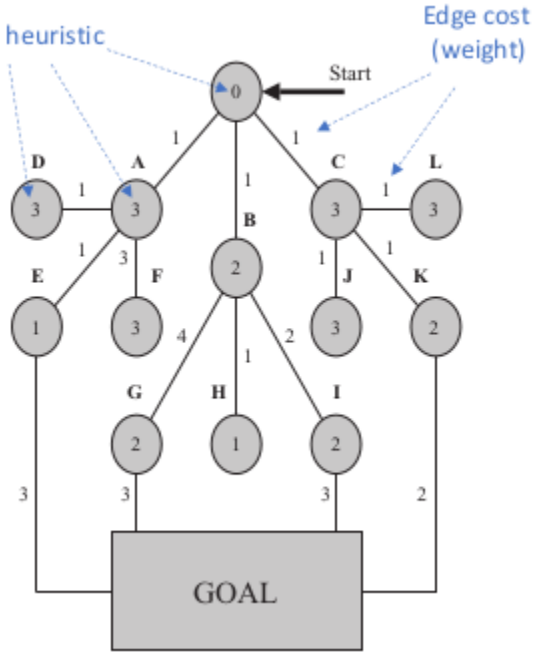
Name: Arpit Savarkar

All the Plots for in this Report can be found at this link : https://github.com/arpit6232/Algorithamic_Motion_planning-/tree/main/PRM_and_RRT/plots

Q1.

Ans:
Problem Set:



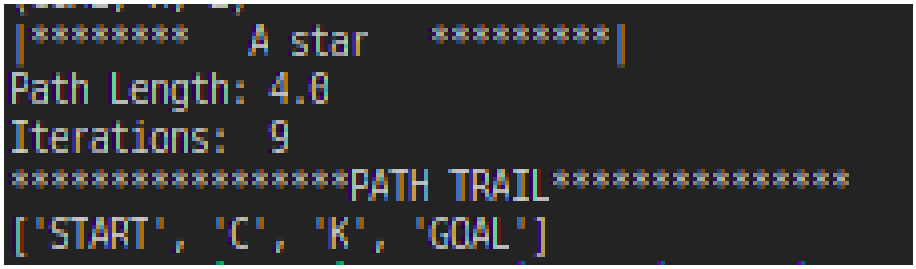
Algorithm 24 A^* Algorithm

Input: A graph

Output: A path between start and goal nodes

repeat
  Pick n_{best} from O such that $f(n_{best}) \leq f(n), \forall n \in O$.
  Remove n_{best} from O and add to C.
  If $n_{best} = q_{goal}$, EXIT.
  Expand n_{best}: for all $x \in \text{Star}(n_{best})$ that are not in C.
  if $x \notin O$ **then**
    add x to O.
  else if $g(n_{best}) + c(n_{best}, x) < g(x)$ **then**
    update x's backpointer to point to n_{best}
  end if
until O is empty

1.a:
Ans: Following Excerpts have been taken from the Code attached undertaken for this question.



1.b:
Ans:

A min heap is a data structure that stores keys and values, and sorts the keys in terms of their associated values from smallest to largest. Dijkstra's algorithm processes vertices with a lower accumulated cost before other ones. Dijkstra's algorithm is largely the same as BFS. Progressing through the vertices, while adding and popping them off of the min heap until the goal vertex is processed. One interesting case however, is if we find a new path to a vertex that is already in the open heap but has not been processed yet. In this case, we have to check if the newly found path to this vertex is cheaper than the old path. If it is, we need to update its cost in the min-heap otherwise, no action is required. Once we process the goal vertex we must have necessarily processed all possible predecessor vertices of the goal node. Since a predecessor must have accumulated distance less than or equal to the goal vertex. Since all predecessor vertices have been processed, we will have found the shortest path to the goal vertex. Once the goal vertex has been processed, so the algorithm can terminate.

Algorithm Dijkstra's(G,s,t)

```
1.  open ← MinHeap()
2.  closed ← Set()
3.  predecessors ← Dict()
4.  open.push(s, 0)
5.  while ! open.isEmpty() do
6.    u, uCost ← open.pop()
7.    if isGoal(u) then
8.      return extractPath(u, predecessors)
9.    for all v ∈ u.successors()
10.     if v ∈ closed then
11.       continue
12.     uvCost ← edgeCost(G, u, v)
13.     if v ∈ open then
14.       if uCost + uvCost < open[v] then
15.         open[v] ← uCost + uvCost
16.         predecessors[v] ← u
17.     else
18.       open.push(v, uCost + uvCost)
19.       predecessors[v] ← u
20.   closed.add(u)
```

Dijkstra's algorithm required us to search almost all of the edges present in the graph, even though only a few of them were actually useful for constructing the optimal path.

A* Star algorithm uses an Admissible Heuristic in addition to path length cost. In Dijkstra's algorithm, we push our open vertices onto a min heap along with their accumulated cost from the origin. The main heap then sorts the open vertices by their associated accumulated cost.

The min difference between Dijkstra's algorithm and A* is that instead of using the accumulated cost, we use the accumulated cost plus Heuristic Cost, the heuristic estimated remaining cost to the goal vertex as the value we push onto the min heap.

The min heap then essentially sorts the open vertices by the estimated total cost to the goal. In this sense, A* biases the search towards vertices that are likely to be part of the optimal path according to our search heuristic.

A* biases the search towards vertices that are likely to be part of the optimal path according to our search heuristic. Since we are storing a heuristic based total cost and the min-heap, we also need to keep track of the true cost of each vertex as well. **The thing to note is that if we take our heuristic to be zero for all vertices which is still an admissible heuristic, we then end up with Dijkstra's algorithm.**

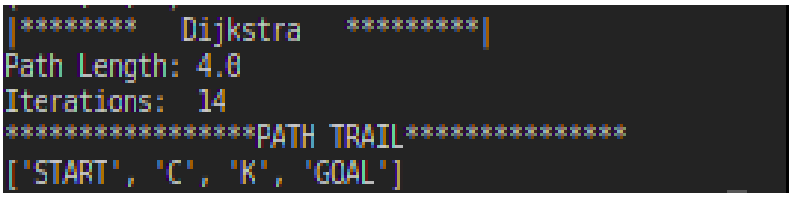
```
1.  if v ∈ open then
2.          if uCost + uvCost + h(v) < open[v] then
3.              open[v] ← uCost + uvCost + h(v)
4.              costs[v] ← uCost + uvCost
5.              predecessors[v] ← u
6.      else
7.          open.push(v, uCost + uvCost)
8.          costs[v] ← uCost + uvCost
9.          predecessors[v] ← u
```

Algorithm A*(G,s,t)

```
1.  open ← MinHeap()
2.  closed ← Set()
3.  predecessors ← Dict()
4.  open.push(s, 0)
5.  while ! open.isEmpty() do
6.    u, uCost ← open.pop()
7.    if isGoal(u) then
8.      return extractPath(u, predecessors)
9.    for all v ∈ u.successors()
10.     if v ∈ closed then
11.       continue
12.     uvCost ← edgeCost(G, u, v)
13.     if v ∈ open then
14.       if uCost + uvCost + h(v) < open[v] then
15.         open[v] ← uCost + uvCost + h(v)
16.         costs[v] ← uCost + uvCost
17.         predecessors[v] ← u
18.     else
19.       open.push(v, uCost + uvCost)
20.       costs[v] ← uCost + uvCost
21.       predecessors[v] ← u
22.   closed.add(u)
```

Implementation Difference specified more clearly.

1.c:
Ans: Following Excerpts have been taken from the Code attached undertaken for this question.



1.d
Ans:
A* Algorithm uses an admissible heuristic along with path cost which optimizes the path search for lower iteration count.

A* Algorithm undertakes lower number of iterations to converge to a goal.

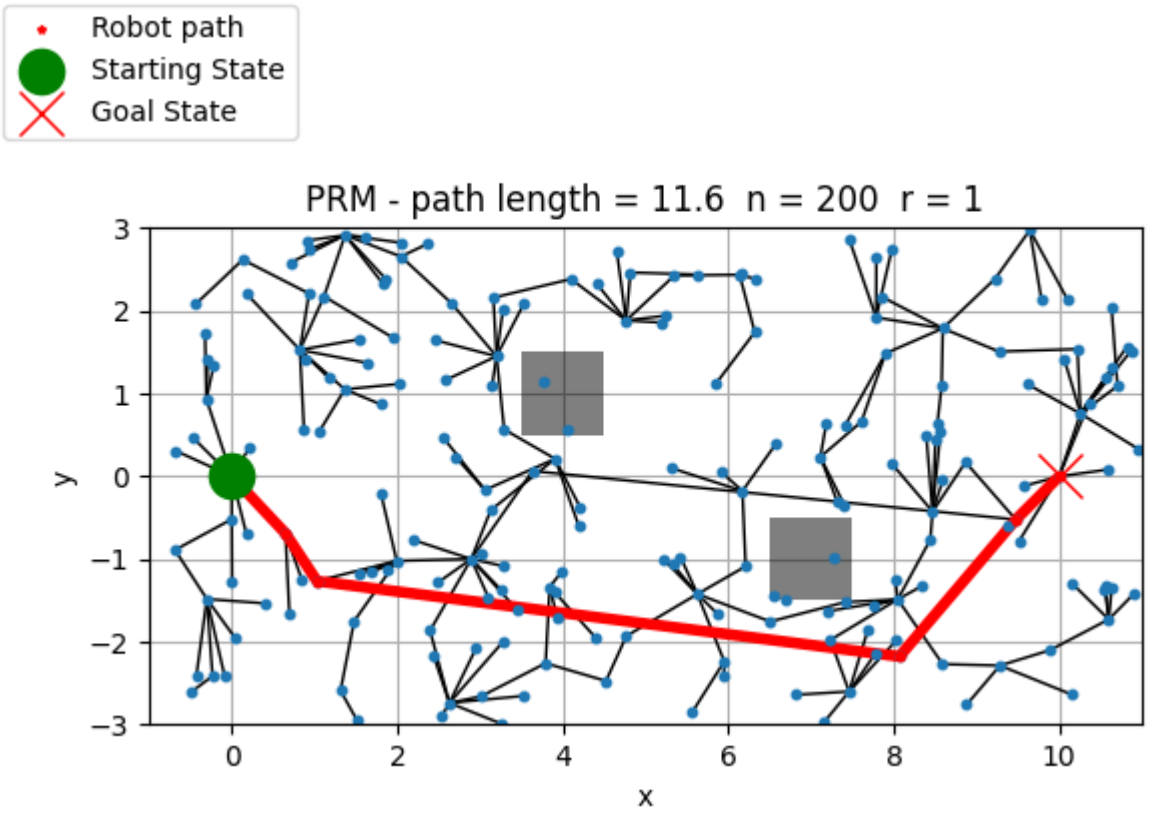
(i) To search a large graph, between goal and Start node, A* algorithm should be used, because it takes lower iteration to optimize the shortest path.

(ii) Between "v" start and every $v \in V$, Dijstra's algorithm should be chosen as it looks at each nodes in relative isolation and may end up finding a path that do not contribute to the overall shortest path.

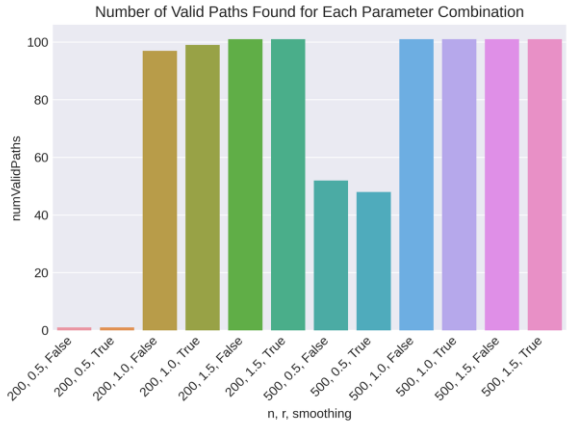
1.e
Ans:

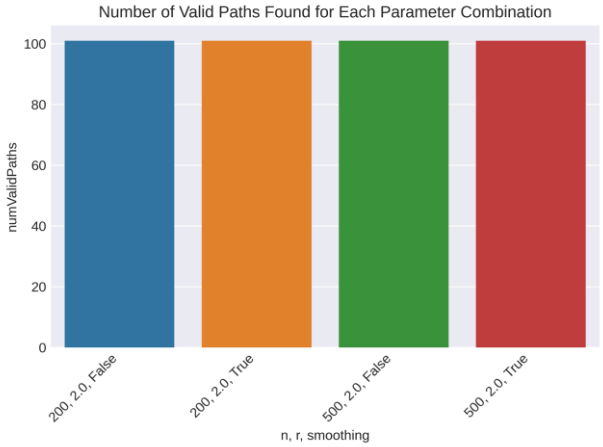
Since undirected graphs have a high probability that the graph search can get stuck in a cyclic loop. An Additional track data structure/variable would be needed to be kept track of parent node. That is the parent node lookup querying will need to be modified to prevent cyclic lock.

Q2:
2.a
Ans:
(i) Plot the roadmap and the solution path for n = 200 and r = 1. Indicate the path length in the title of the plot.
Ans:

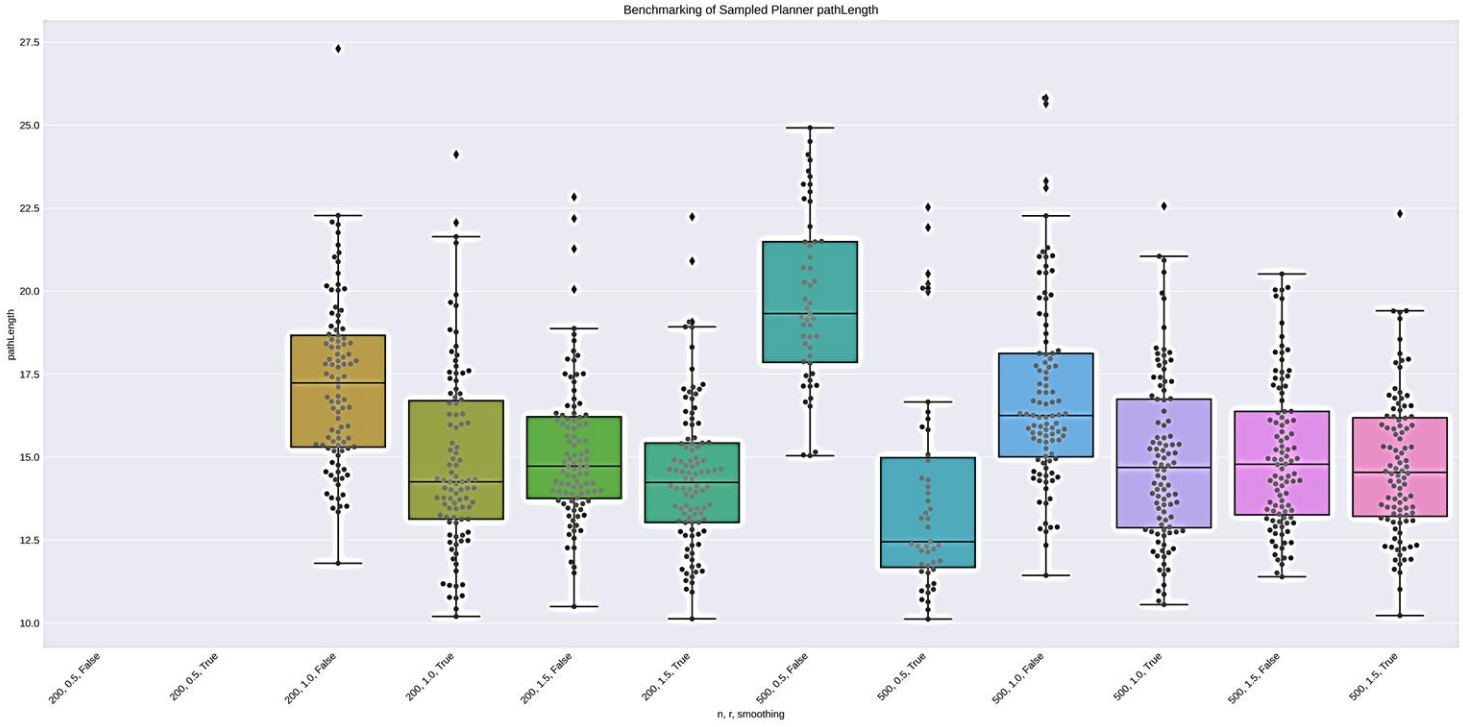


(ii) Vary n and r and benchmark your solutions in three categories of number of valid solutions, path length, and computation time. For benchmarking, use 100 runs for each
(n, r) ∈ {(200, 0.5), (200, 1), (200, 1.5), (200, 2), (500, 0.5), (500, 1), (500, 1.5), (500, 2)}.
Show your results using boxplots.
Ans:
Due to some unavoidable computer crashes, I had plot the box plot results for radius = 2, separately.

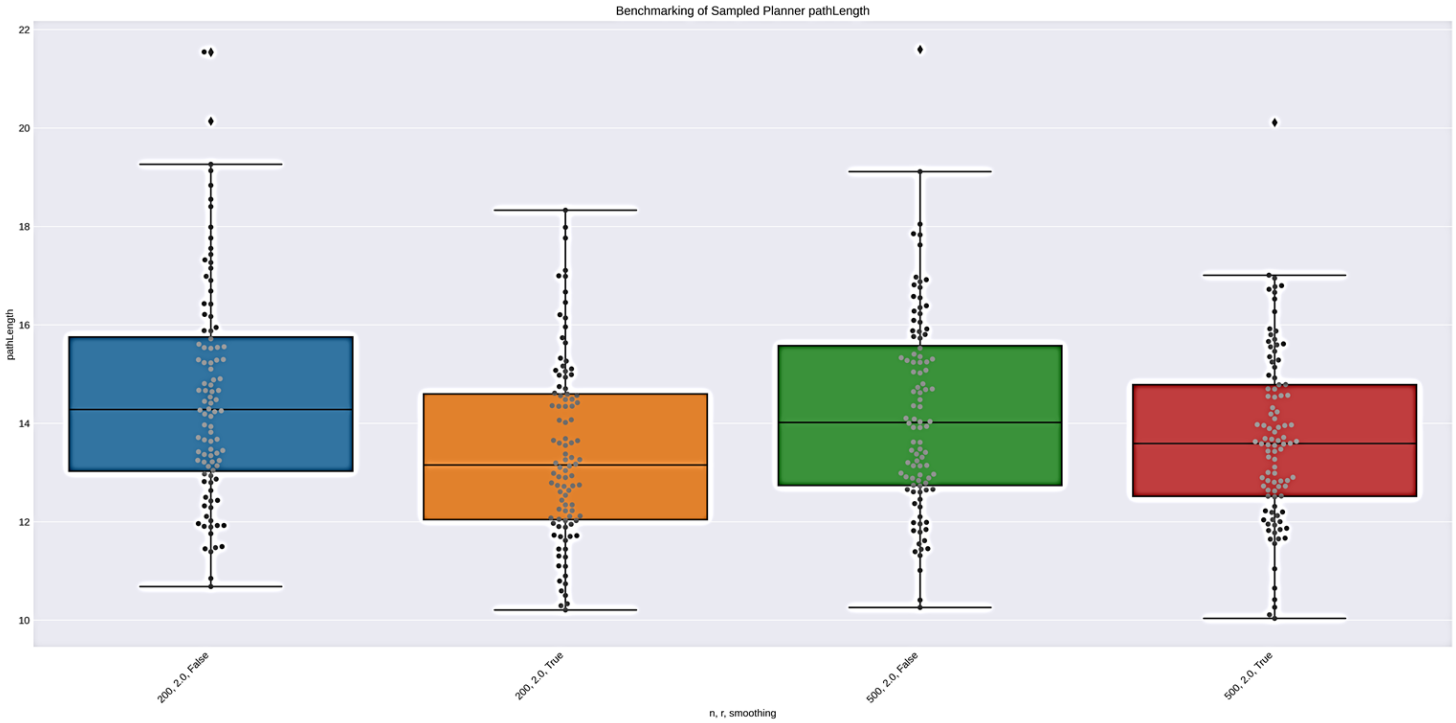




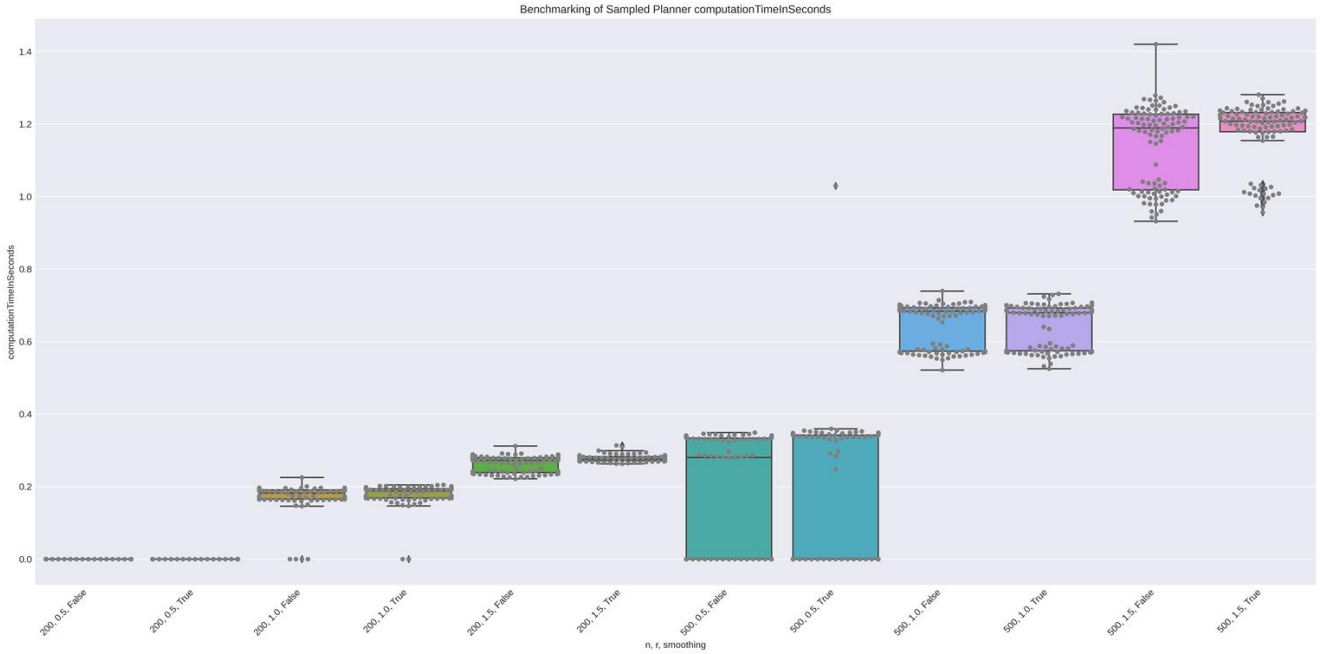
Benchmarking Analysis – Path Length :



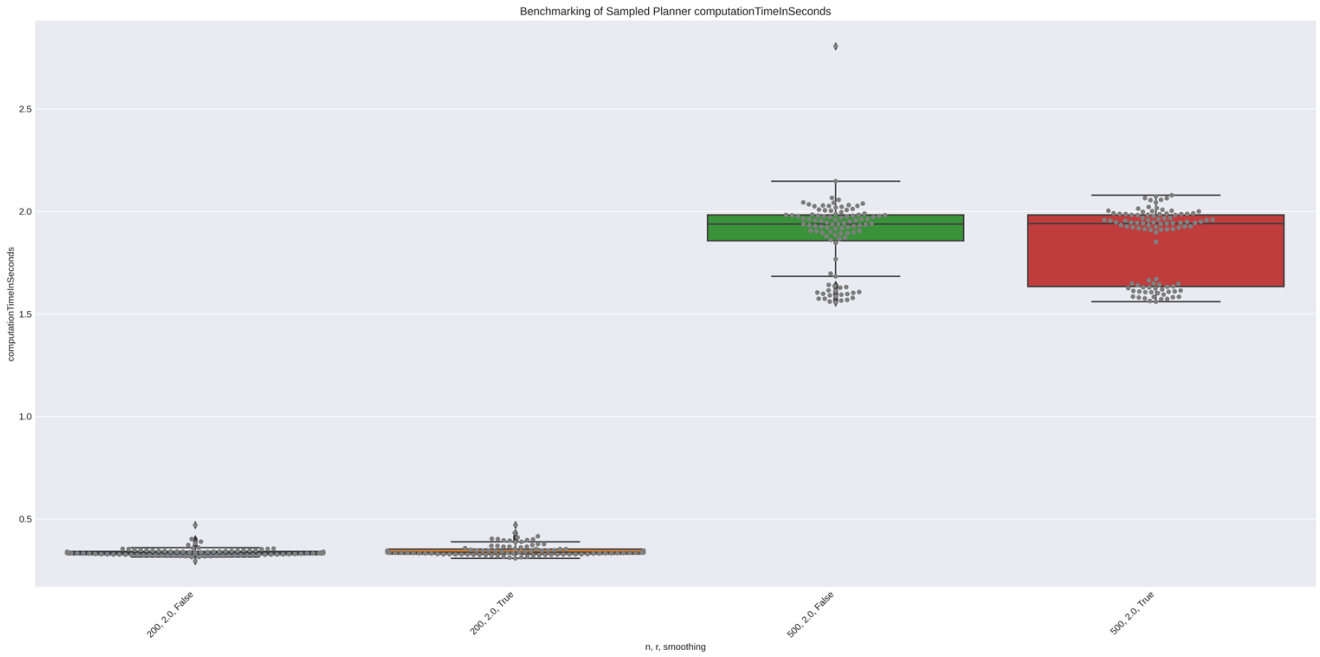
Radius =2



Benchmarking Analysis – Computation Time



Radius = 2



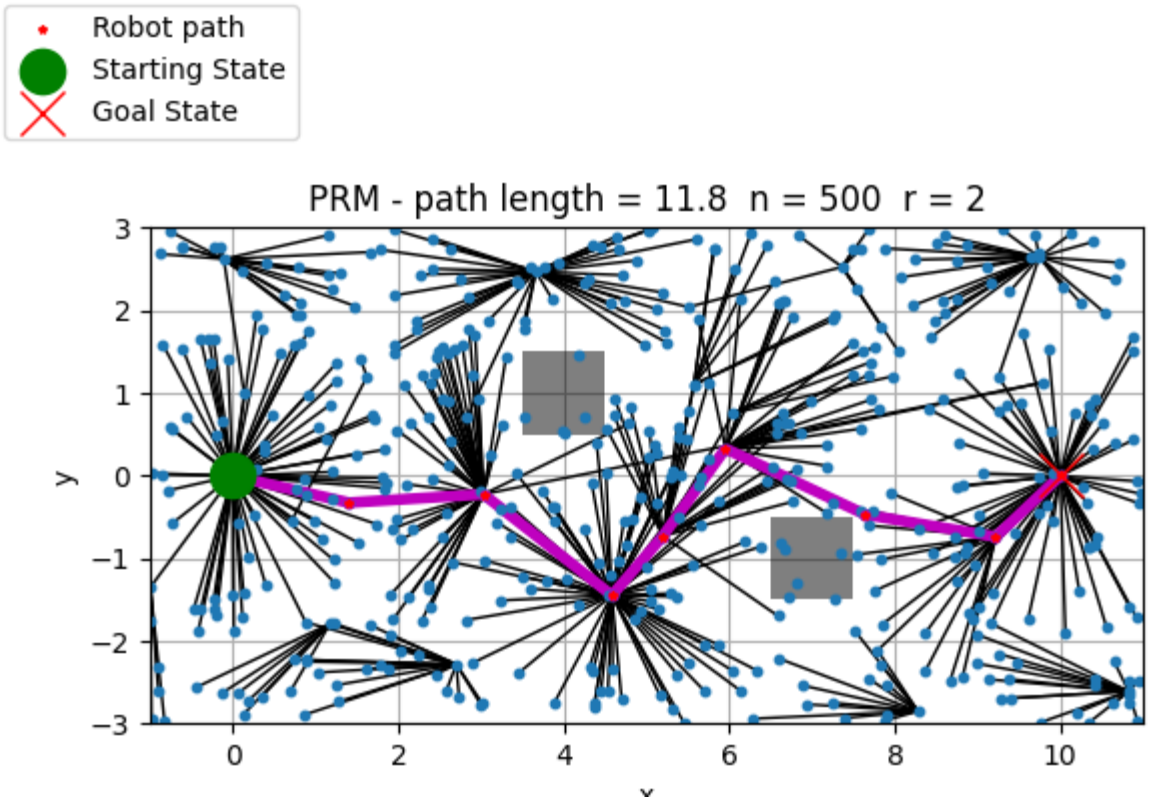
(iii) Based on your empirical evaluations, what are the optimal values for n and r? Justify your answer.

Ans: I found the results to be optimum for Sample Size of 200 Units and radius of 2.0, with Smoothing (Even better) which seems to be a good balance between computation time and path length for the PRM algorithm, which uses A* algorithm under the hood. For such a small workspace, 200 units seem to populate the graph uniformly with sufficient number of edges. For higher number of nodes, there would be other possible paths, but at the cost of more compute time. A radius of 2.0 ensures that the number of edges on the graph do not lead to convoluted creation of path lengths. Thus a higher number of both nodes and radius would take more time for the graph search for a optimal path. Combined with more number of nodes, if the edge lengths are longer, it is even observed that the path lengths are oscillatory. Too small number of sample node, would lead to a edge configurations, that do not lead to goal node as indicated by the successful path length plot.

(iv) Augment your PRM planner with a path smoothing technique and re-evaluate your benchmarks. What are the optimal values for n and r with path smoothing? Justify your answer.

Ans:

With Path smoothing, I would again stick to the same combination, but the results are close {n=500, r=1.0 with Smoothing} are comparable to {n=200, r=2.0, with smoothing}. Lead to comparable results, but I would still prefer lower number of samples on the graph to gain for the smaller compute time. For this particular workspace. Which does not have a large number of obstacles. Path Smoothing does reduce the path length upto an extent.


Robot path
Starting State
Goal State
PRM - path length = 11.8 n = 500 r = 2
y
x
3
2
1
0
−1
−2
−3
0
2
4
6
8
10

Q2b)

(i) Plot the roadmap and the solution path for n = 200 and r = 2. Indicate the path length in the title of the plots.

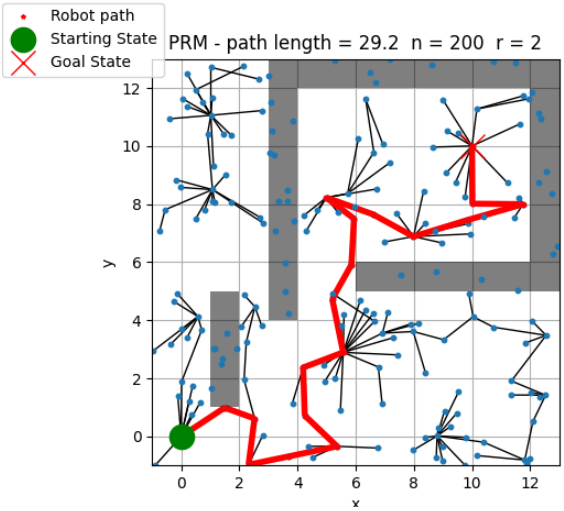


Figure 2: Successful Path Found for the node and radius configuration

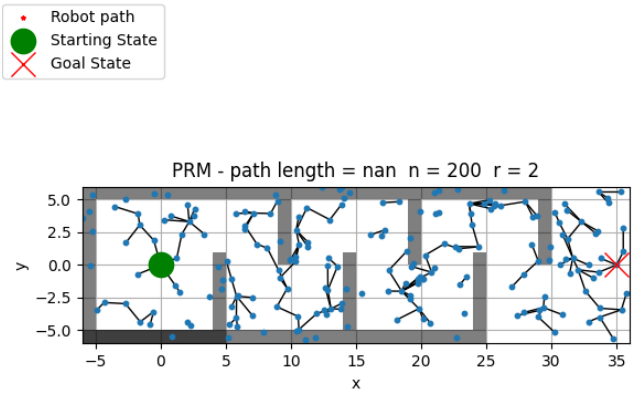
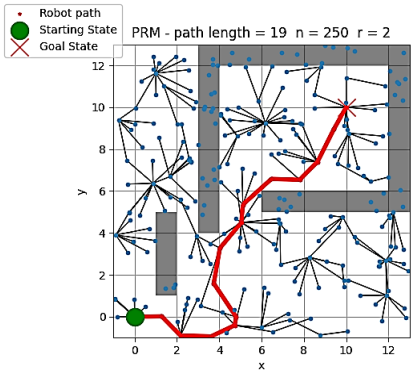


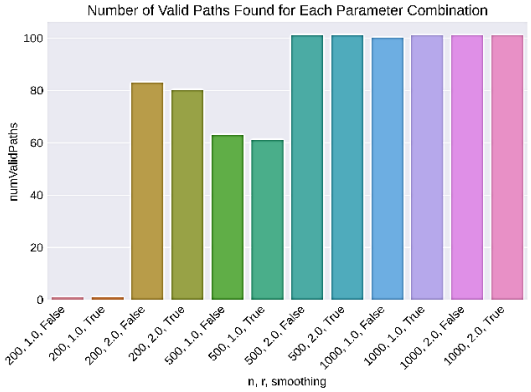
Figure 1 NO Successful Path Found for the node and radius configuration

(ii) Vary n and r and benchmark your solutions in three categories of number of valid solutions, path length, and computation time. For benchmarking, use 100 runs for each (n; r) 2 f(200; 1); (200; 2); (500; 1); (500; 2); (1000; 1); (1000; 2):

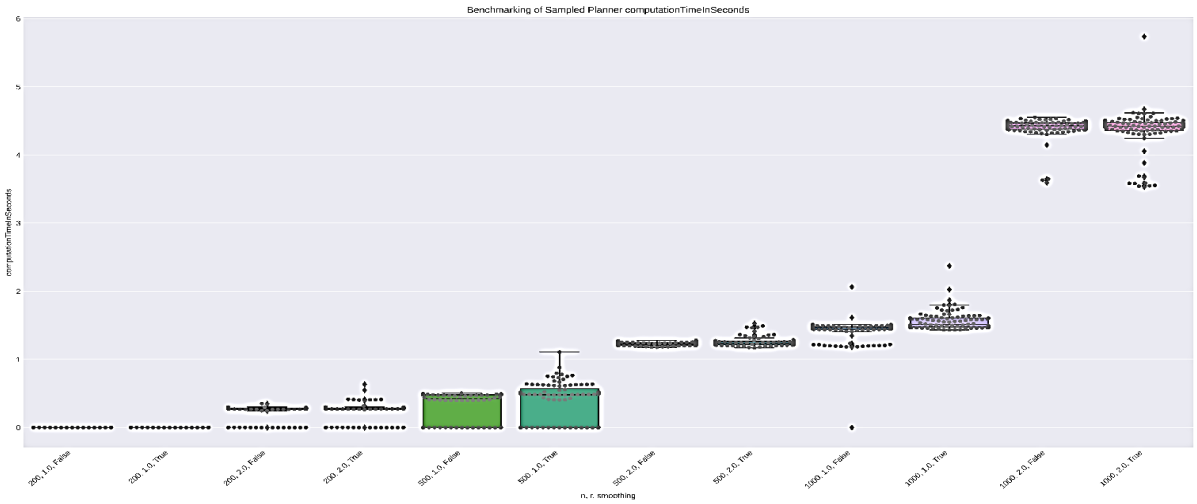
Ans:

Workspace (5 Obstacles)

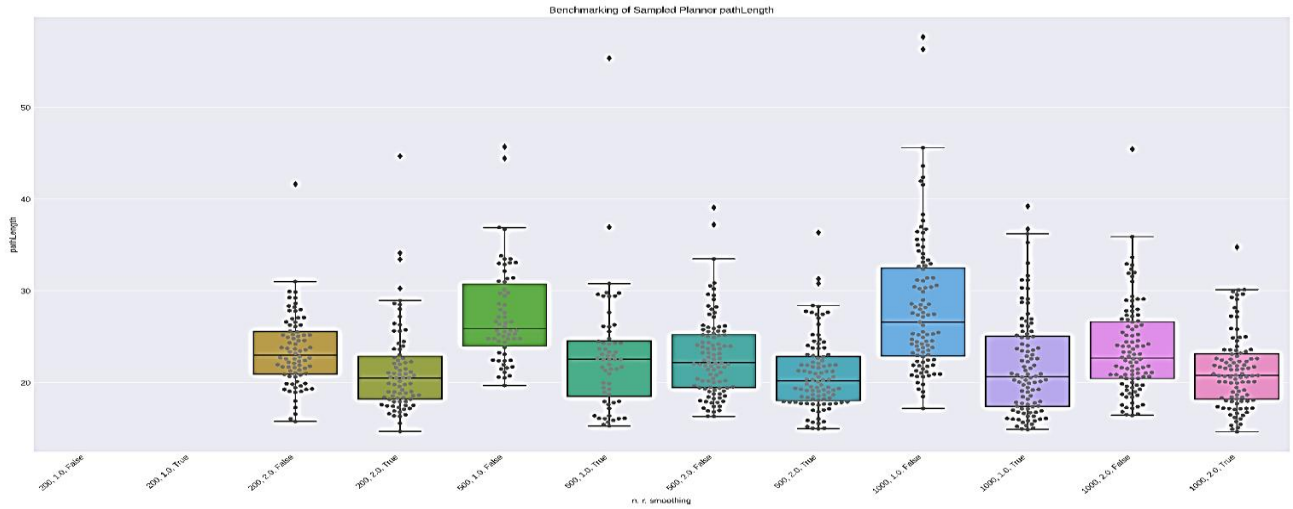




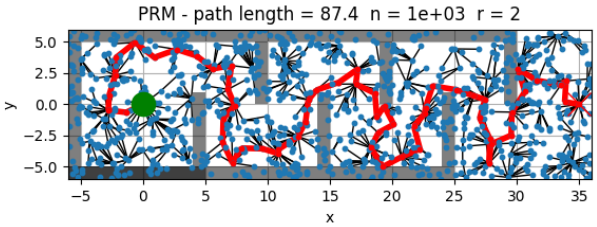
Benchmarking Analysis – Computation Time

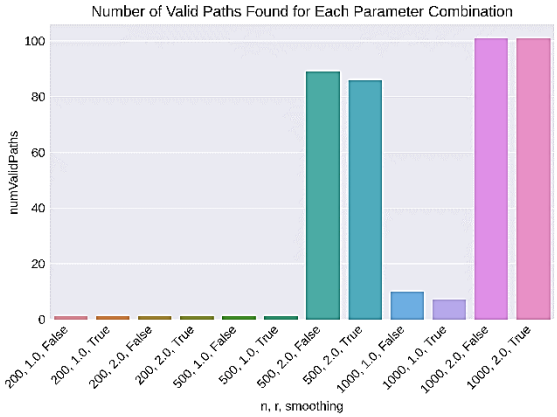


Benchmarking Analysis – Path Length

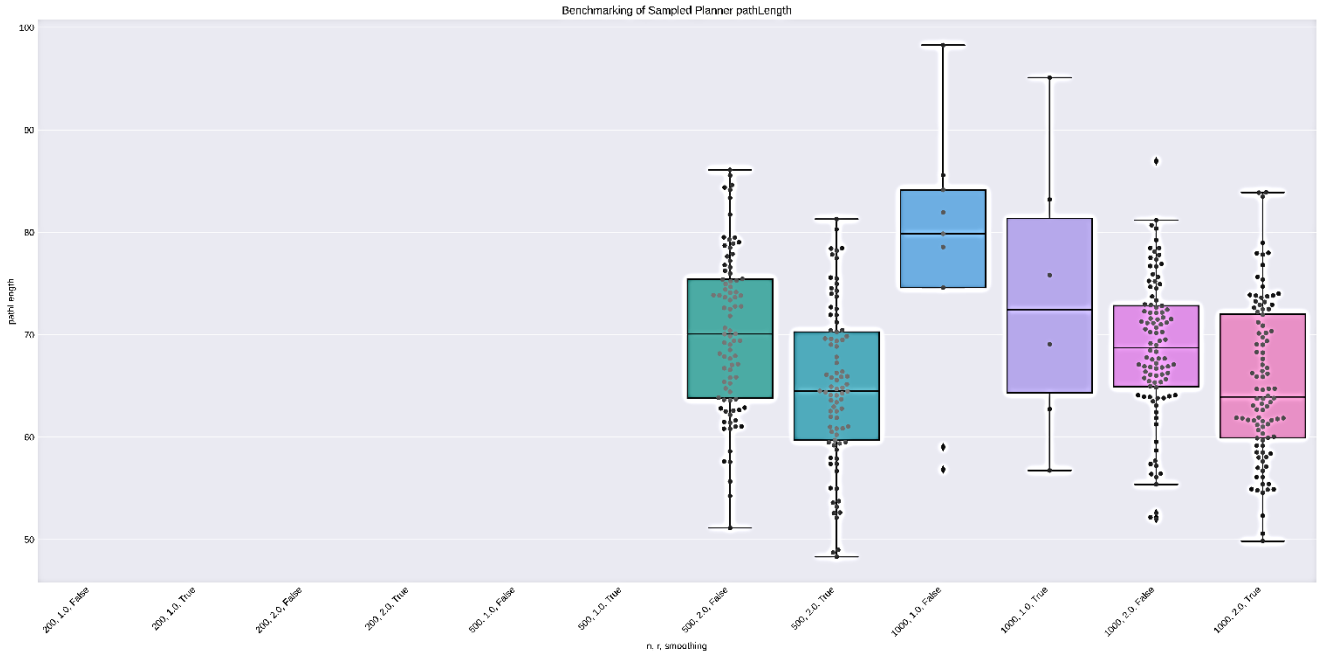


Workspace 2(9 obstacles)

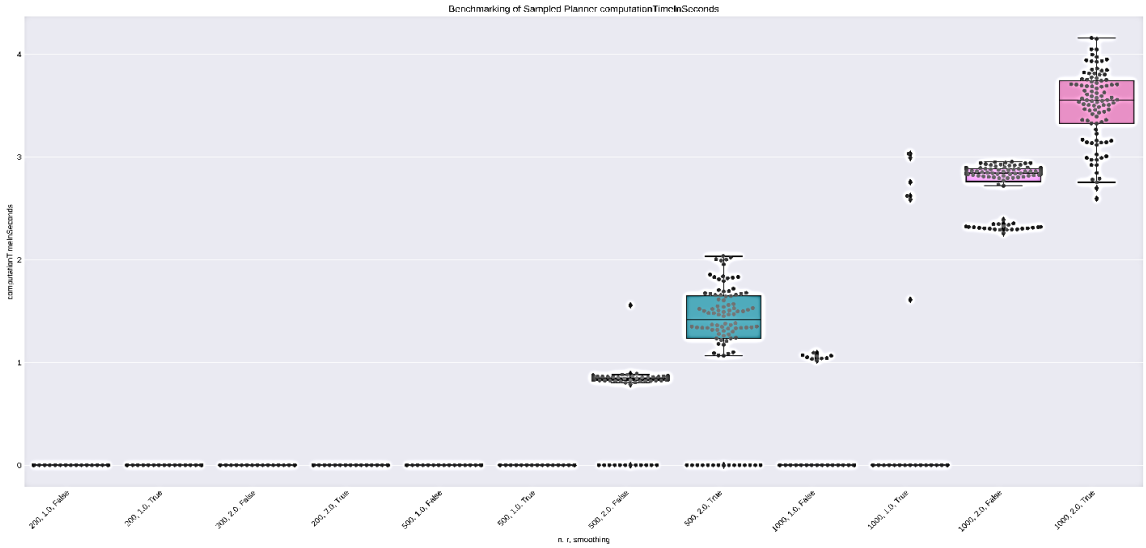




Benchmarking – Path Length



Benchmarking – Computation Time



(iii) Based on your empirical evaluations, what are the optimal values for n and r in W_1 and W_2? Justify your answer.

Ans:

If we need the to find a parameter that would satisfy both the workspaces with the same parameters, I would consider the number of nodes to be 1000 with radius 2, which assures 100% find path.

For Workspace with 5 obstacles (like a square) .
500 node samples with radius = 2 units, brings 100% path finding guarantees, with the lowest computation time. For all the node sizes smaller than 200 and radius 1, we have no successful paths. But if we are happy with not 100% convergence guarantees, then lower sample nodes (with radius >1), lead to a lower computation time. Although, this workspace is more complex to the previous subsection, more nodes with the same edge length ensures optimal connectivity of graph, for 1000 nodes and radius 2. More edges mean both oscillatory / zig-zag based behaviour. Having too few nodes means lack of edges to goal. For convergence guarantees, we need a large enough radius to get us through the narrow corridors

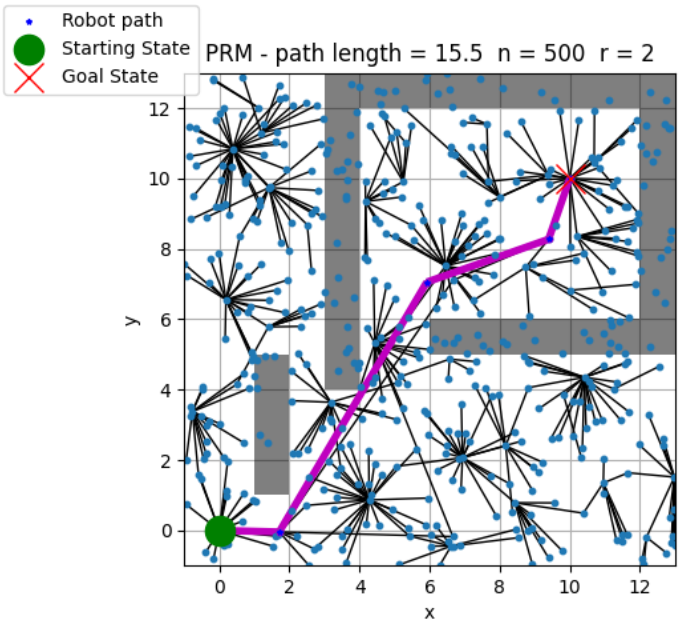
OPTIMUM Sizes: Node sample 500, radius = 2

For Workspace with 9 obstacles, (like a cave tunnel).
Very poor performance is observed, radius<2. With Exception being sample node size of 500,(with the logic being lack of connectivity, between the goal node and the random node generated), which also does not guarantee 100% path finding.

ONLY, sample size of 1000 and radius 2 should be used for 100% path finding guarantees.

(iv) Enable path smoothing option in your PRM and re-evaluate your benchmarks. What are the optimal values for n and r with path smoothing? Justify your answer.

For Workspace with 5 obstacles (like a square) .
Again, I would stick with **Sample size 500, and radius 2** to maintain 100% path finding guarantees, but according to my plots, I can see the smoothing curves, leads to increase in the variance in box plots, but **decrease in computation time** so that would be preferred.

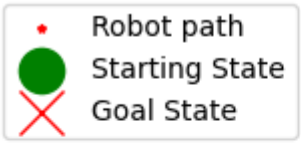


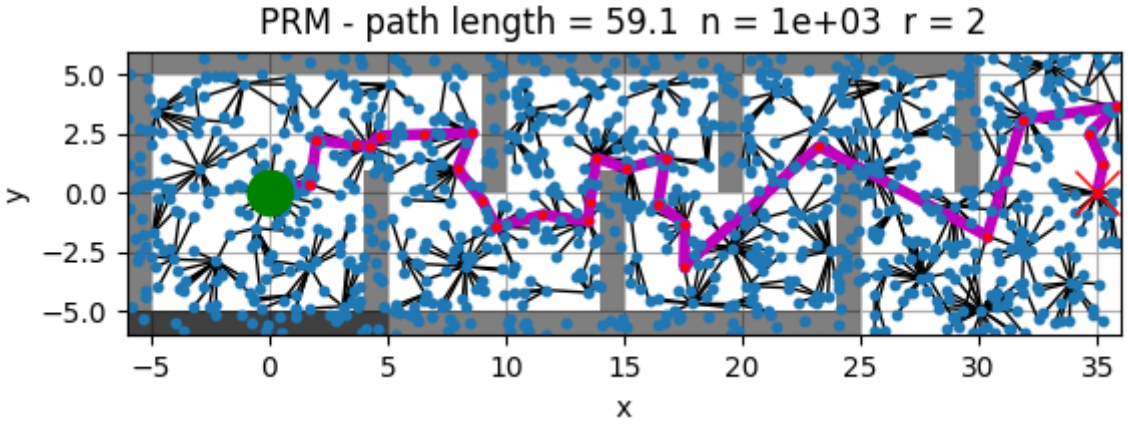
Example of a Plan with Smoothing

For Workspace with 9 obstacles, (like a cave tunnel).

Again to maintain, 100% oath finding guarantees, I would consider **Sample Node size of 1000 and Radius 2**. If my objective is to optimize path length, I would consider smoothing. But if comparative benchmarking analysis for computation time seems to increase a lot with smoothing than without smoothing.

Example of a Plan with Smoothing

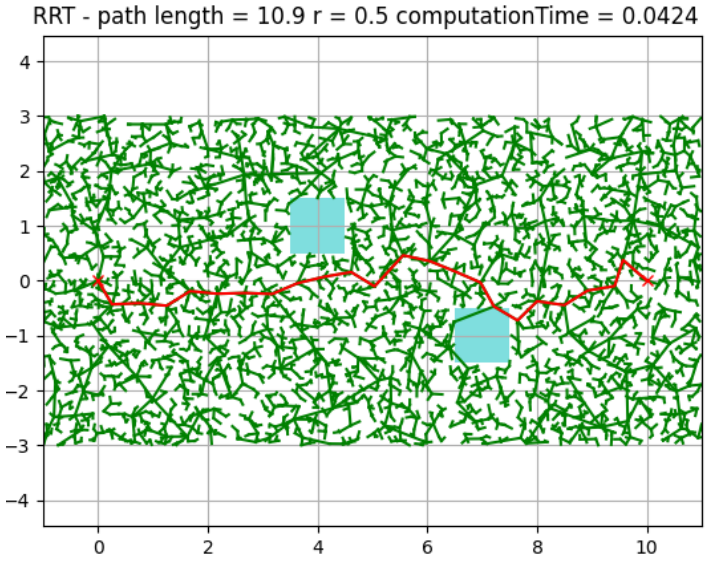


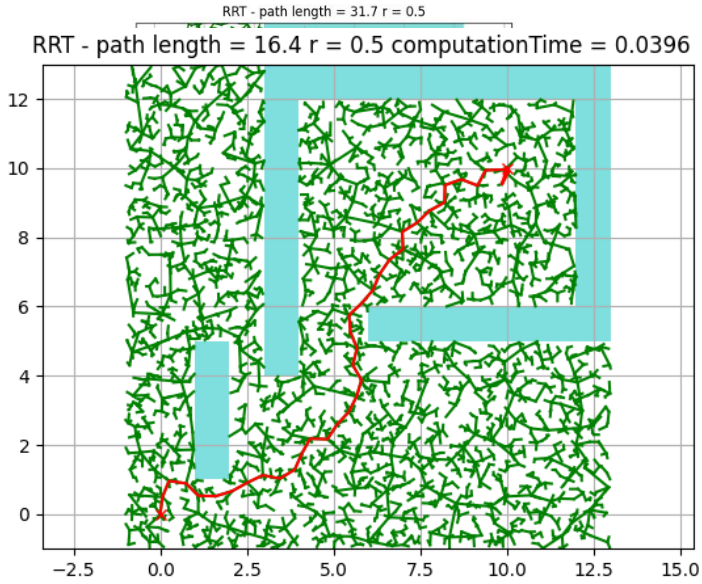


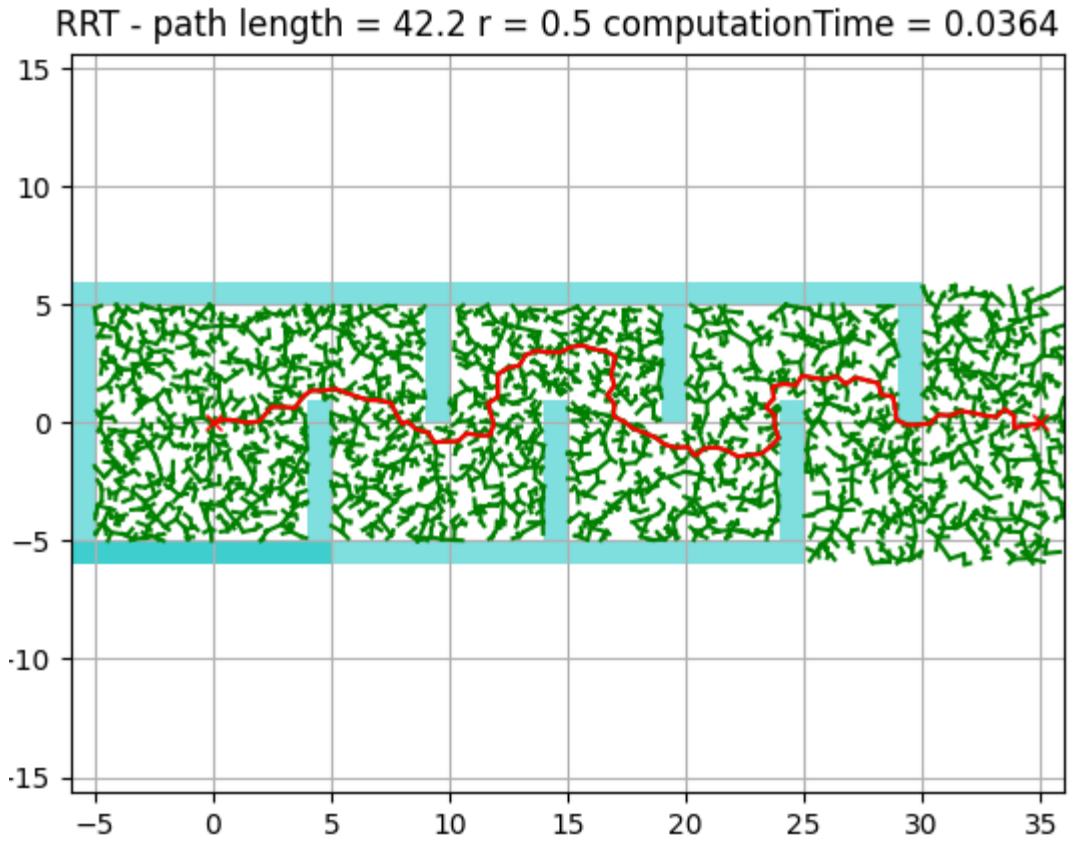
Q2c

Ans: For Manipulators, the C-space has no left and right extreme boundaries for the graph (0 – 2*pi), we need to construct the C-space for the robot and then run our planner in the Learning Phase. We do also need to change up the algorithm to satisfy what was stated above. But that in the graph construction for PRM only. The core PRM algorithm (avoiding obstacles) would not need not change, but Physically to implement the algorithm PRM – SMOOTHING would be necessary to prevent weird path generations. We can use Forward Kinematics to map back the robot into its workspace. If a C-space is expansive, then a roadmap can be constructed efficiently with good connectivity and coverage. **That is Performance of PRM depends on the properties of the C-space. Thus Number of samples and radius will need to be tuned.**

Q3
(a)
Ans:
Number of iterations n = 5000, Computation Time in SECONDS



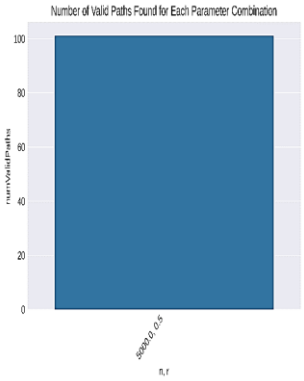


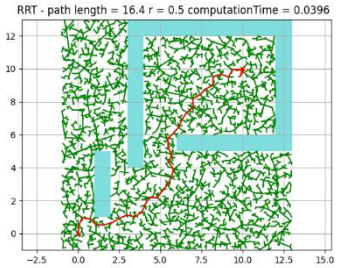


Q3b.
Ans:
(b)
Workspace:

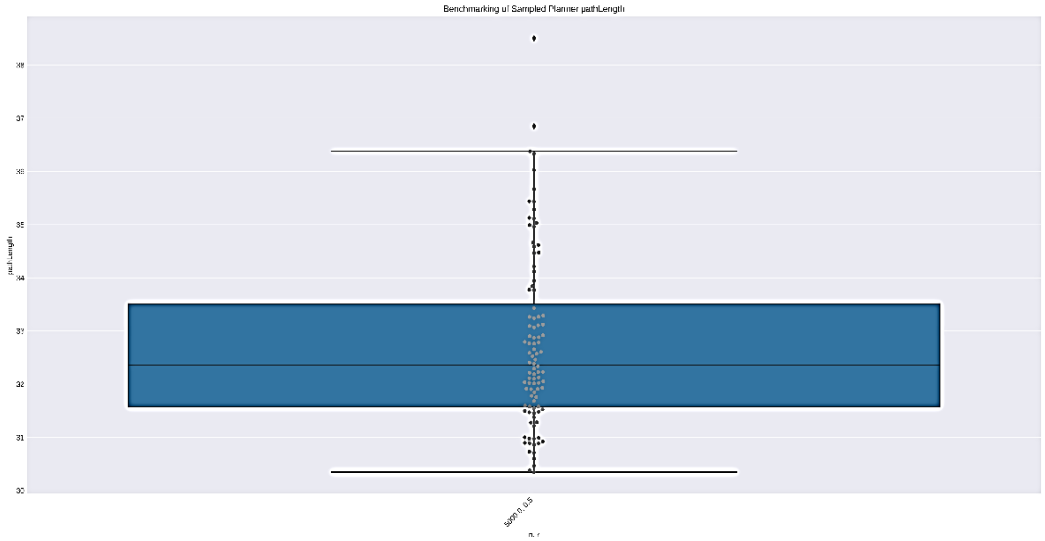
(I) The Benchmarking Analysis shows that RRT always leads to a path from the start node to the goal node as seen below.

Number of Valid Paths Found

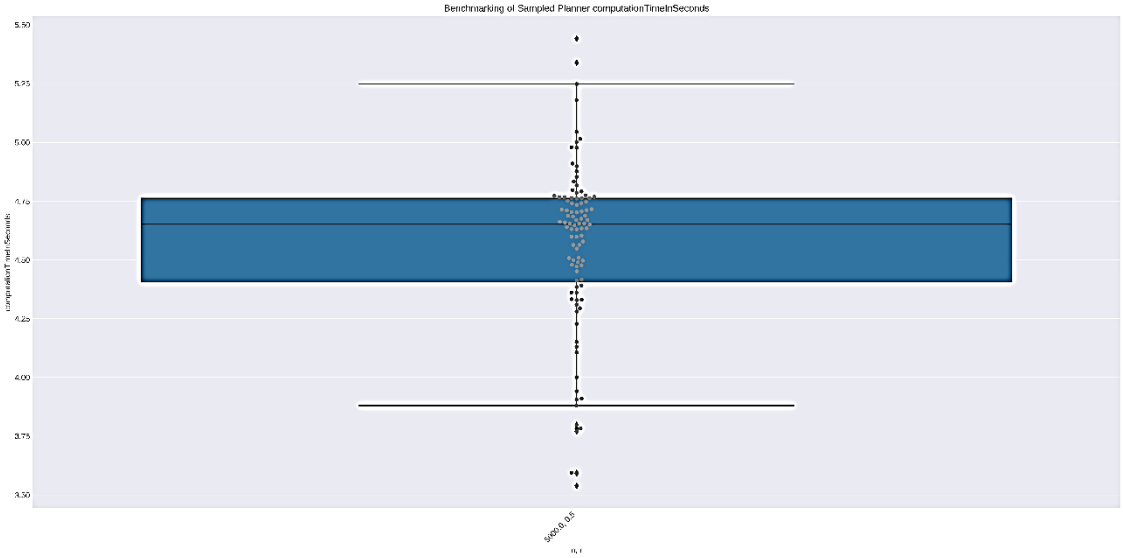




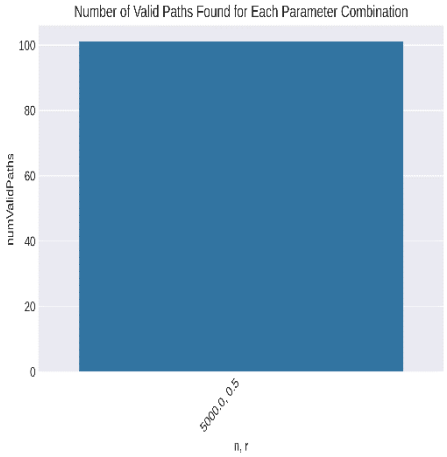
Benchmarking – Path Length

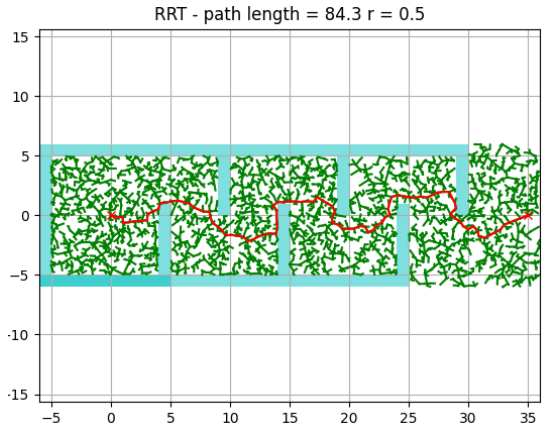


Benchmarking – Computation Time

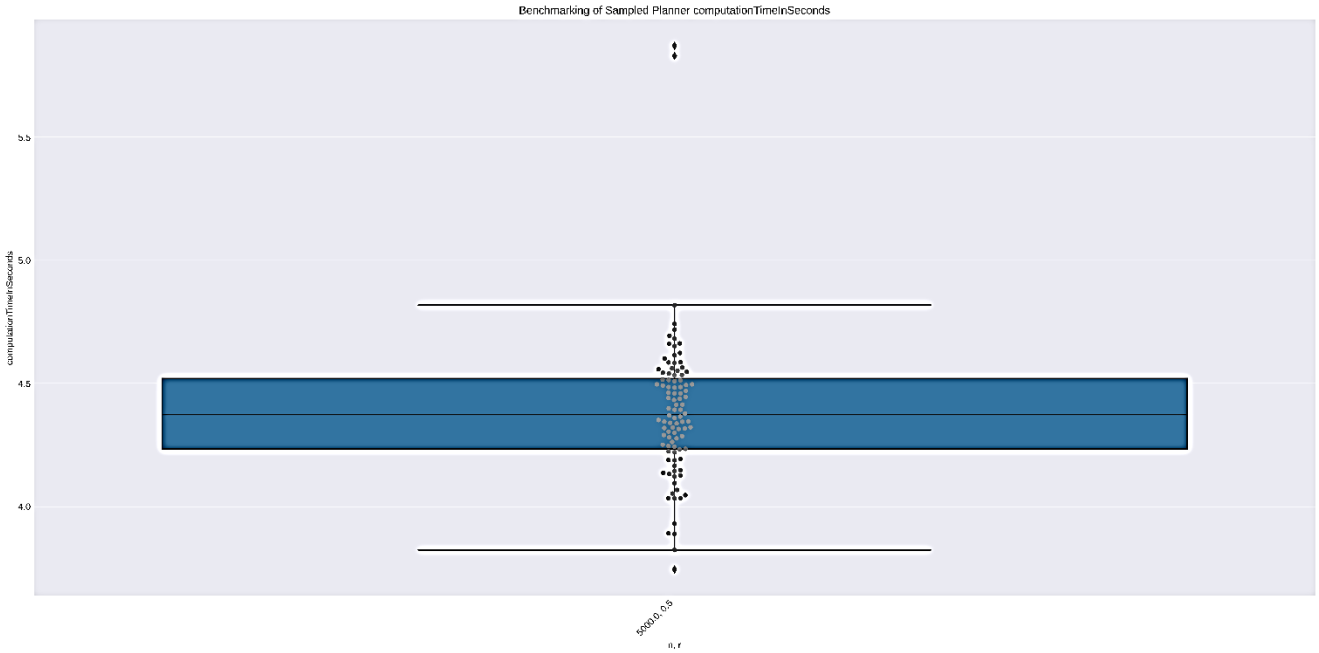


Workspace (II):

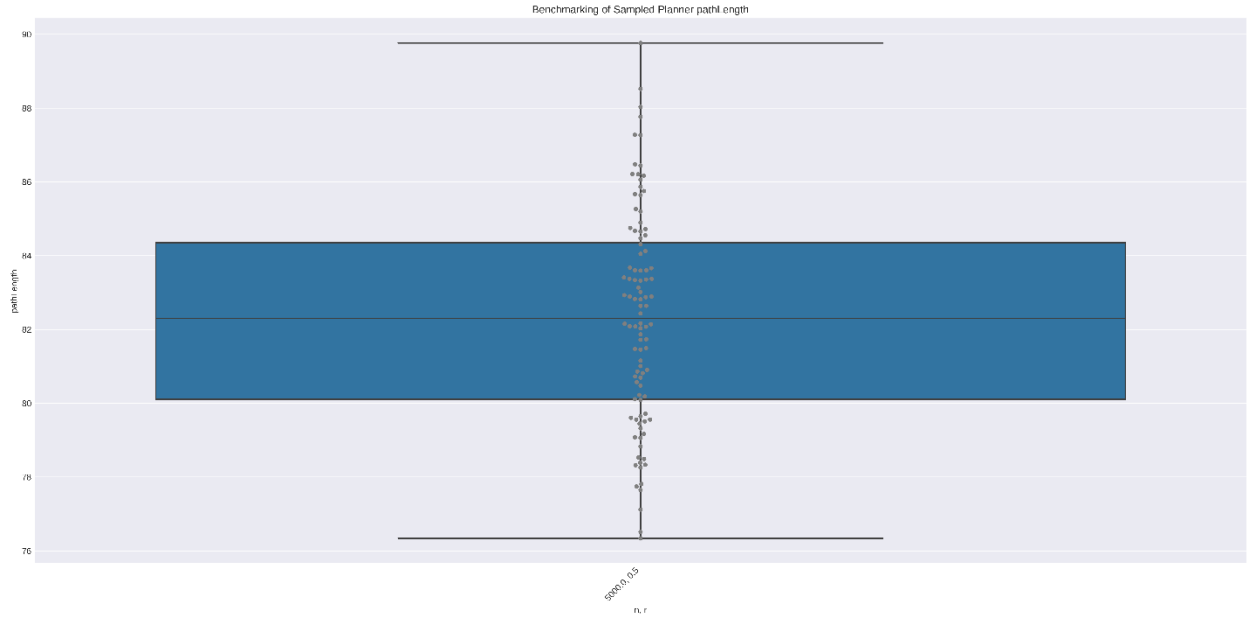




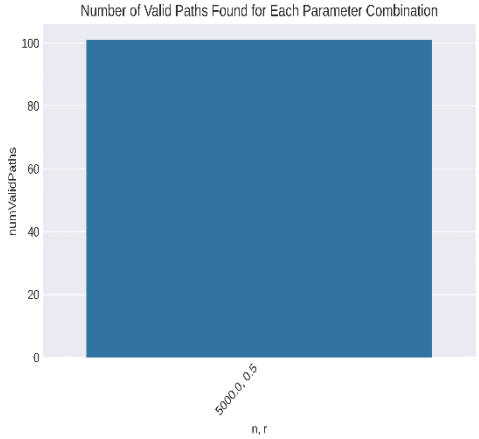
Benchmarking –Computation Time

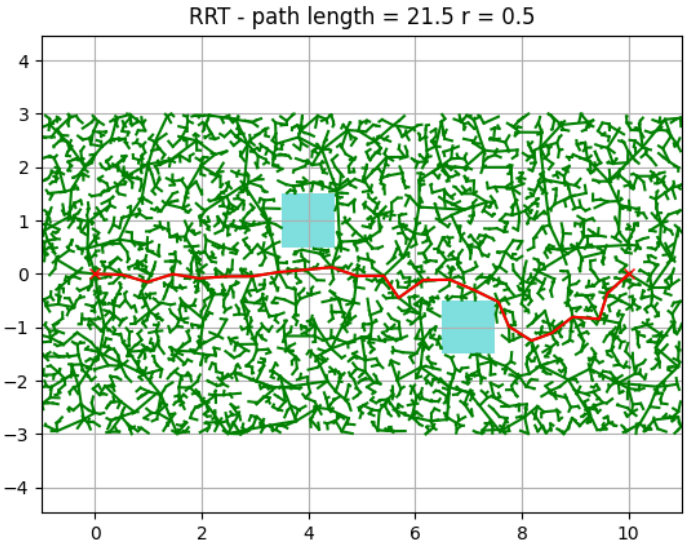


Benchmarking – Path Length

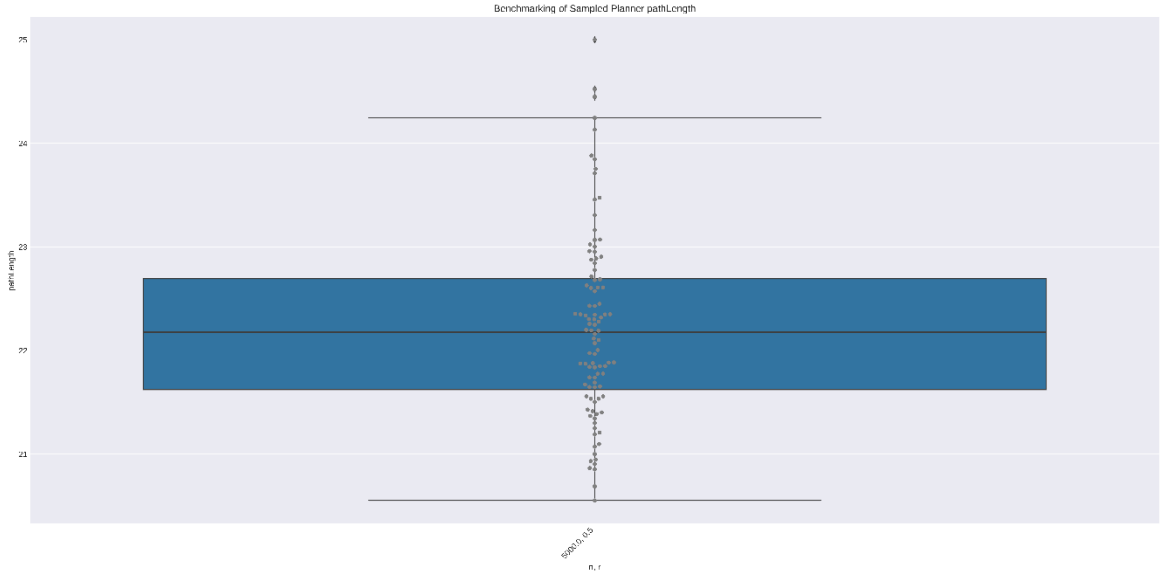


Workspace (III)

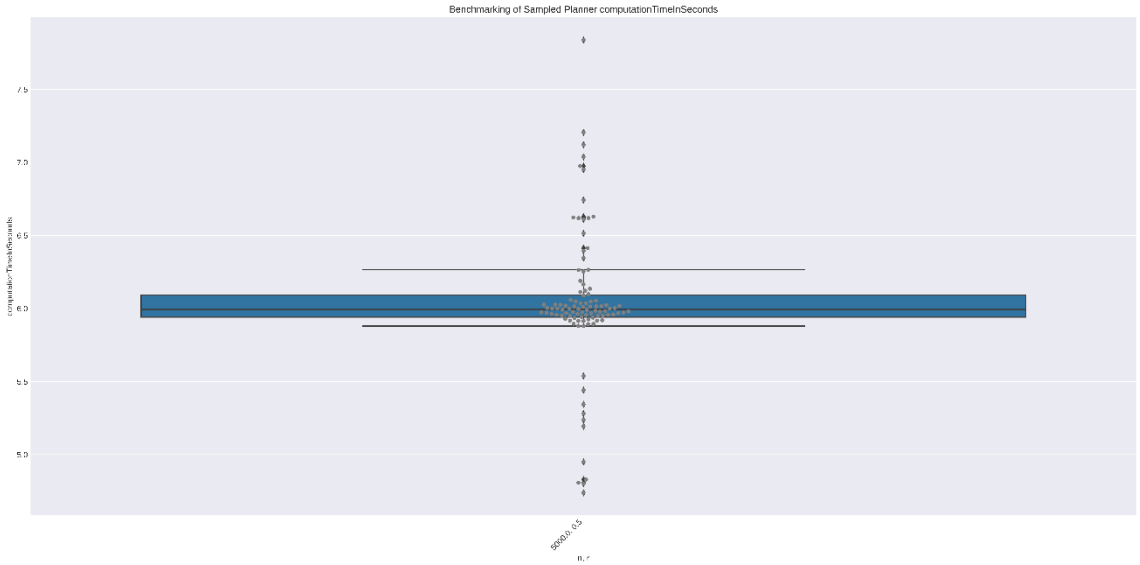




Benchmarking – Path Length



Benchmarking – Computation Time



Q3c:
Ans:
For Manipulators, the C-space has no left and right extreme boundaries for the graph (0 – 2*pi), we need to construct the C-space for the robot and then run our planner in the Learning Phase. We do also need to change up the algorithm to satisfy what was stated above. But that in the graph construction for RRT only. The core RRT algorithm (avoiding obstacles) would not need not change, but Physically to implement the algorithm for optimized other performance other variants of RRT should be used. We can use Forward Kinematics to map back the robot into its workspace.

Q4
Ans:
Gradient Descent with Potential Functions:

Workspace	Computation Time (secs)	Path Length
2 obstacles	7.31	10.59
5 obstacles	11.09	23.64
9 obstacles	17.89	61.54

Gradient Descent along Potential Function is not a trivial task. This algorithm approach is prone to local minima. Chattering Problems also do exist in the Vanilla Implementation of the Gradient Descent in Potential Functions due to cancellations of repulsive and attractive potentials. Additionally, due to large number of hyper-parameters to tune, implementation can take time. Basically, the construction of ideal potential fields could be probably difficult and time consuming. Furthermore, in Higher dimensional workspaces, we might need to sample robot configurations, and the related transforms

On the Positive side, potential field ensures that the robot always moves in the direction of the goal, and do not require post-processing techniques to optimize the solution for better path.
Navigation potential function
• resolves the local minima problem
• difficult to construct
• may be possible in sphere space
• require hand tuning of parameters to find a balance between local minima and non-isolated critical points
• can map star-shaped spaces to sphere spaces

Challenge in using potential functions:
• Non-Euclidean C-space hard to construct
• Use relationship between forces in workspace and C-space
• Define potential functions in workspace and map to C-space
• Use gradient decent for motion planning.

Wavefront Algorithm

Workspace	Computation Time (secs)	Path Length
2 obstacles	0.52	10
5 obstacles	1.27	20
9 obstacles	3.78	44

Wavefront algorithms work in discretized space. This algorithm is complete with respect to the grid. Wave-front algorithm is optimal with respect to the grid. It was observed since that the finer the grid, the robot can close in very near the obstacle, which in the real world can be dangerous.
The advantage of working in the grid world is that it can easily be extended to higher dimensions.

- The grid is an approximation of the space, so the gradient is an approximation of the actual distance gradient.
- Can be generalized into higher dimensions
- Can become computationally intractable in higher dimensions.
- Discretization-based approach and uses only Attractive Potential. (removes obstacles)

PRM:

Workspace	Computation Time (secs)	Path Length
2 obstacles	0.25	11.8
5 obstacles	0.6	15.55
9 obstacles	1.6	58.6

Probabilistic Roadmap are a probabilistically complete algorithm, if a path exists, the probability of not finding it $\rightarrow 1$ as number of samples $\rightarrow \infty$. PRM-based planners aim to construct a roadmap that captures the whole connectivity of the configuration space. Performance of sampling-based planners depend on the properties of the C-space.

- This algorithm sacrifices optimality for completeness even though there are techniques to make the search itself more optimal. Once the PRM is constructed, efficient graph search algorithms can be em-ployed to query through the node to find the find path.
- PRM can sometimes be difficult to find path through narrow corridors.
- To optimize for path length, post-processing steps are necessary, vanilla implementation of PRM does not give optimal path, smoothing would be needed to use.
- Constructing PRM also helps us find path not just from start to end goal but from any node to other.
- We also extend the same graph for a different start and goal node. Thus PRM are very efficient and easy to implement algorithms.

RRT:

Workspace	Computation Time (secs)	Path Length
2 obstacles	0.0451	12.1
5 obstacles	0.0351	15.6
9 obstacles	0.0358	42.21

Rapidly Exploring Random tree for the obstacles and workspaces, we are working in we get shorter path lengths and faster time convergence. Adding more samples to the C-space can create more optimum path

lengths for the graph which is not the case in PRM which samples node all over the C-Space. Thus time taken to converge, is shorter in comparison to PRM.

1. BasicRRT, takes only one small step when adding a new branch.
   (i) Tree is pulled towards random directions based on the uniform sampling of Q.
   (ii) RRT relies on nearest neighbours and distance metric to grow trees towards the node.
   (iii) RRT adds Voronoi bias to tree growth space.
2. RRT suffers from the following :
   (i) Large number of configurations, increases computational cost.
   (ii) It becomes increasingly difficult to guide to guide the tree towards previously unexplored parts of the free configuration space .
   (iii) Metric sensitivity is found.