RBE550: Motion Planning

Project 2: Barking up a Random Tree

Student 1: Dhruv, Agrawal
Student 2: Shreyas, Khobragade

Project goals:

- To implement a simple sampling-based motion planner in OMPL to plan for a rigid body.
- To systematically compare this planner to existing methods in the library.

1 Theoretical Questions

1. **Write two key-differences between the Bug 1 and Bug 2 algorithms.**

Bug 1	**Bug 2**
Motion to goal path is a straight line path from current position of the point robot to goal	Motion to goal path is a fixed straight line (m-line) from the start position of the point robot to goal position.
Object following strategy is implemented in Bug 1 by following the entire perimeter of the obstacle and then deciding on the nearest point to goal position from the entire perimeter. Bug 1 performs an exhaustive search to find the optimal leave point. When the obstacles are complex Bug 1 performs better.	The robot switches from obstacle following strategy to goal following strategy as soon as it comes to a point on the m-line which is closer to goal than the previous time the robot crossed the m-line. Bug 2 performs an opportunistic/greedy search. When the obstacles are simple then Bug 2 performs better.

Table 1: Key differences between Bug 1 and Bug 2 algorithms

2. **A heuristic h is admissible if**

$$h(n) \leq T(n)$$

where T is the true cost from n to the goal, and n is a node in the graph. In other words an admissible heuristic never overestimates the true cost from the current state to the goal.
A stronger property is consistency. A heuristic is consistent if for all consecutive states n, n'

$$h(n) \leq T(n, n') + h(n')$$

where T is the true cost of node n to its adjacent node n'.

- **Imagine that you are in the grid world and your agent can move up, down, left, right, and diagonally, and you are trying to reach the goal cell $f = (g_x, g_y)$. Define an admissible heuristic function h_a and a non-admissible heuristic h_b in terms of x, y, g_x, g_y. Explain why each heuristic is admissible/non-admissible.**

The figure shows the graphical representation of the problem statement.

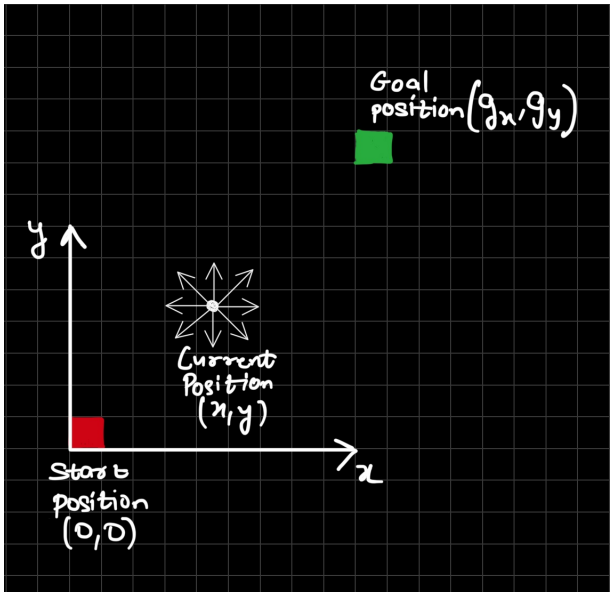


Figure 1: Robot and goal representation

An admissible heuristic for the problem can be:

$$min(|g_x - x|, |g_y - y|)$$

To prove this, let us consider some cases:
Case 1:
Let us assume that y-coordinates of the current and goal positions are same. This means that the goal is horizontally to the left/right of the robot.
In this case the heuristic is $min(|g_x - x|, 0) = 0$.
The actual cost is $T = (|g_x - x|)$.
Since the heuristic is less than actual cost, is it thus admissible.
Case 2:
Let us assume that x-coordinates of the current and goal positions are same. This means that the goal is vertically above/below the robot.
In this case the heuristic is $min(|g_x - x|, 0) = 0$.
The actual cost is $T = (|g_y - y|)$.
Since the heuristic is less than actual cost, is it thus admissible.
Case 3:
Let us assume that the difference between x-coordinates of the current and goal positions is equal to the y-coordinates of the current and goal positions. This means that the goal is $|g_x - x|$ or $|g_y - y|$ diagonal steps away from the robot (Assuming 1 horizontal/vertical step cost is the same as 1 diagonal step cost equal to 1 unit).
In this case the heuristic is $min(|g_x - x|, |g_y - y|) = |g_x - x| = |g_y - y|$.
The actual cost is also $T = |g_x - x| = |g_y - y|$.
Since the heuristic is equal to the actual cost, is it thus admissible.
Case 4:
Now let us assume that the robot is at some random non-symmetric point from the goal as shown below:

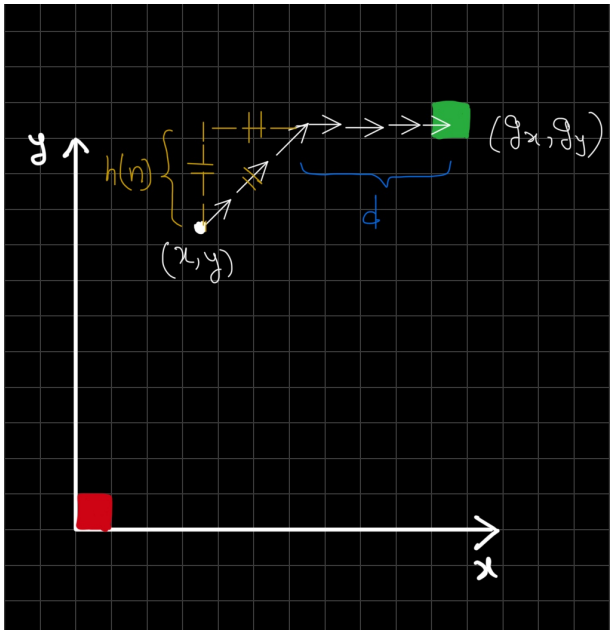


Figure 2: Heuristic vs True cost

From Figure 2 we can see that, the cost associated when the robot moves in the diagonal is equal to the heuristic cost.
The true cost is the **heuristic cost + some distance d** in the x or y direction.
Finally, in the worst case, the heuristic cost will be equal to the true cost when d=0 but never less than the true cost.
A non-admissible heuristic for the problem can be:

$$2 \cdot (|x - g_x| + |y - g_y|)$$

this is double the Manhattan distance To prove this is inadmissible, we only need to prove it false for one case.

Let us assume that the difference between x-coordinates of the current and goal positions is equal to the y-coordinates of the current and goal positions. This means that the goal is $g_x - x$ or $g_y - y$ diagonal steps away from the robot (Assuming 1 horizontal/vertical step cost is the same as 1 diagonal step cost equal to 1 unit).
In this case the heuristic is

$$2 \cdot (|x - g_x| + |y - g_y|) = 2 \cdot |g_x - x| + 2 \cdot |g_y - y|$$

The actual cost is also $T = |g_x - x| = |g_y - y|$.
We can clearly see that the heuristic is far greater than the actual cost, is it thus inadmissible.

- **Let h_1 and h_2 be consistent heuristics. Define a new heuristic $h(n) = max(h_1(n), h_2(n))$. Prove that h is consistent.**

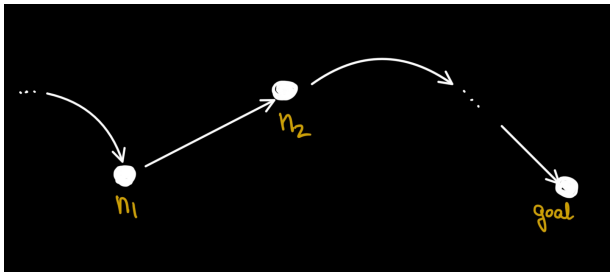


Figure 3: Graphical path

We have two heuristics h1 and h2.
Referring to the Figure above we can assume that, node n_1 is near to the goal node than node n_2.

This means that:

$$h_1(n_2) \leq h_1(n_1)$$
$$h_2(n_2) \leq h_2(n_1)$$

We take a new heuristic $h = max(h_1, h_2)$.
Case 1:

$$h_2(n_1) > h_1(n_1)$$
$$h_2(n_2) > h_1(n_2)$$

In this case it is simple to prove that h is consistent -

$$h(n_1) = max(h_1(n_1), h_2(n_1)) = h_2(n_1)$$
$$h(n_2) = max(h_1(n_2), h_2(n_2)) = h_2(n_2)$$

Since h2 is consistent

$$h_2(n_2) \leq h_2(n_1) \leq h_2(n_2) + T(n_1, n_2)$$

Substituting $h_2(n_2) = h(n_2)$ and $h_2(n_1) = h(n_1)$ in the above equation, we get -

$$h(n_2) \leq h(n_1) \leq h(n_2) + T(n_1, n_2)$$

This is the condition for consistency, which implies that h is consistent.
Similarly we can prove consistency of h if $h_1(n_1) > h_2(n_1)$ and $h_1(n_2) > h_2(n_2)$.
Case 2:

$$h_2(n_1) > h_1(n_1)$$
$$h_2(n_2) < h_1(n_2)$$

We calculate the h heuristics:

$$h(n_1) = max(h_1(n_1), h_2(n_1)) = h_2(n_1)$$
$$h(n_2) = max(h_1(n_2), h_2(n_2)) = h_1(n_2)$$

Since h1 and h2 are consistent

$$h_2(n_2) \leq h_2(n_1) \leq h_2(n_2) + T(n_1, n_2)$$
$$h_1(n_2) \leq h_1(n_1) \leq h_1(n_2) + T(n_1, n_2)$$

Substituting $h_1(n_2) = h(n_2)$ and $h_2(n_1) = h(n_1)$ in the above equations, we get -

$$h_2(n_2) \leq h(n_1) \leq h_2(n_2) + T(n_1, n_2)$$
$$h(n_2) \leq h_1(n_1) \leq h(n_2) + T(n_1, n_2)$$

From our assumption that $h_2(n_1) > h_1(n_1)$

$$h(n_1) > h_1(n_1) \geq h(n_2)$$

or we can write it as -

$$h(n_1) >= h(n_2)$$

From our assumption that $h_2(n_2) < h_1(n_2)$, we can write it as -

$$h_2(n_2) < h(n_2)$$

Adding $T(n_1, n_2)$ on both sides we get

$$h_2(n_2) + T(n_1, n_2) < h(n_2) + T(n_1, n_2)$$

Substituting this above we can write:

$$h_2(n_2) \leq h(n_1) \leq h(n_2) + T(n_1, n_2)$$

Since from above

$$h(n_1) >= h(n_2)$$

Substituting this above we get

$$h(n_2) \leq h(n_1) \leq h(n_2) + T(n_1, n_2)$$

This is the condition for consistency, which implies that h is consistent.
Similarly we can prove consistency of h if $h_1(n_1) > h_2(n_1)$ and $h_1(n_2) < h_2(n_2)$.
Thus for any relation between h_1 and h_2 we can prove consistency of h.

3. **Consider workspace obstacles A and B. If $A \cap B \neq \Phi$, do the configuration space obstacles QA and QB always overlap? If $A \cap B = \Phi$, is it possible for the configuration space obstacles QA and QB to overlap? Justify your claims for each question.**
   For Part 1 of the question:
   Yes, if $A \cap B \neq \Phi$,this means that the objects are in collision in workspace.
   If the objects are in collision in workspace, this means that there is a region in the Configuration space where both the objects are present. Within that configuration space region the robot is in collision with both the objects.
   For Part 2 of the question:
   Yes, even if the objects are not in collision in the workspace, they can be in collision in the Configuration space. This is possible when the two obstacles are close to each other such that the space in between them is smaller than the size of the robot.

4. **Suppose you are planning for a point robot in a 2D workspace with polygonal obstacles. The start and goal locations of the robot are given. The visibility graph is defined as follows:**

   - **The start, goal, and all vertices of the polygonal obstacles compose the vertices of the graph.**
   - **An edge exists between two vertices of the graph if the straight line segment connecting the vertices does not intersect any obstacle. The boundaries of the obstacles count as edges.**
     - **Provide an upper-bound of the time it takes to construct the visibility graph in big-O notation. Give your answer in terms of n, the total number of vertices of the obstacles. Provide a short algorithm in pseudocode to explain your answer. Assume that computing the intersection of two line segments can be done in constant time.**
     - **Can you use the visibility graph to plan a path from the start to the goal? If so, explain how and provide or name an algorithm that could be used. Provide an upper-bound of the run-time of this algorithm in big-O notation in terms of n (the number of vertices in the visibility graph) and m (the number of edges of the visibility graph). If not, explain why.**

   For Part 1 of the question:
   The upper-bound of time to construct the visibility graph is $\mathbf{O(n^3)}$.
   To construct a visibility graph, for each vertex $v \in V$, where $V = \{ v_1, v_2, .., v_n \}$ is a set of vertices of polygons in the configuration space including the start and goal configurations, we must find which other vertices are visible to v. The naive way to achieve this is to test all line segments $\overline{vv_i}$, $v \neq v_i$ to check if they intersect an edge of any polygon. If a pair of points do not intersect with any edge, and that the pair is external to the obstacles, they are visibility edges. Otherwise, they are not. For a particular $\overline{vv_i}$, there are $\mathbf{O(n)}$ intersections to check because there are $\mathbf{O(n)}$ edges from the obstacle. Now, there are $\mathbf{O(n)}$ potential segments emerging from v, therefore for a particular v, there are $O(n^2)$ tests to find which vertices are visible from v. And, this must be checked for all $v \in V$ and hence, the upper-bound of time to construct a visibility graph will be $\mathbf{O(n^3)}$.

Algorithm 1 Computing Visibility Graphs of Obstacles
Input: a set of points defining the obstacles, start and goal.
Output: a set of points which correspond to the visibility edges of the obstacles

```
function constructVisibilityGraph(obstacles, start, goal)
    vertices ← extractVerticesfromObstacles(obstacles)   ▷ Extract all vertices from the given obstacles
    add(start, goal) to vertices                          ▷ Add the start and goal points to the list of vertices
    visibilityGraph ← initializeGraph()                   ▷ Initialize the visibility graph

    for i = 0 to length(vertices) - 1 do
        for j = i + 1 to length(vertices) do
            v1 ← vertices[i]                              ▷ Get the first vertex
            v2 ← vertices[j]                              ▷ Get the second vertex
            if isVisible(v1, v2, obstacles) then          ▷ Check if v1 is visible from v2
                addEdge(visibilityGraph, v1, v2)          ▷ If visible, add an edge between them in the graph
    return visibilityGraph                                ▷ Return the constructed visibility graph

function isVisible(v1, v2, obstacles)                     ▷ Check visibility between two vertices
    for each edge in obstacles do
        if doIntersect(v1, v2, edge) then  ▷ If the line segment between v1 and v2 intersects an obstacle edge
            return false                                  ▷ They are not visible to each other
    return true                                           ▷ They are visible to each other

function doIntersect(v1, v2, edge) ▷ Implementation of segment intersection check (assumed to be O(1))
▷ This function checks if the line segment from v1 to v2 intersects with the given edge
```

For Part 2 of the question:
Yes, you can use a visibility graph to plan a path from a start point to a goal point in a 2D environment. This can be done as follows,

- First, create a visibility graph from the obstacles in the environment. Each vertex corresponds to a point of interest, and edges represent direct line-of-sight connections between those points.
- Identify your start point and goal point. These may or may not be directly part of the vertices in the graph.
- If the start or goal points are not part of the existing vertices, we can add them and connect them to any vertices they have direct visibility to.
- Finally, use a pathfinding algorithm to find the shortest path from the start vertex to the goal vertex.

Pathfinding algorithms such as Dijkstra's or A* algorithm can be used to find the path from start to goal.

The runtime of Dijkstra's algorithm will be $\mathbf{O}((\mathbf{m}+\mathbf{n})\log\mathbf{n})$, where n is the number of vertices and m is the number of edges. Setting up the initial distances takes $\mathbf{O}(\mathbf{n})$ time, extracting the minimum distance for each of the n vertices from the queue takes $\mathbf{O}(\mathbf{n}\log\mathbf{n})$ time and relaxing an edge involves updating distances and may require re-inserting vertices, as each edge is processed once when checking neighboring vertices, taking $\mathbf{O}(\mathbf{m}\log\mathbf{n})$ time. Combining all this, the overall complexity comes out to be $\mathbf{O}((\mathbf{m}+\mathbf{n})\log\mathbf{n})$.

2 Programming Exercises

(a) **Fillout the missing functions in CollisionChecking.cpp by implement collision checking for a point robot within the plane, and a square robot with known side length that translates and rotates in the plane in. There are many ways to do collision checking with a robot that translates and rotates. Here are some things to consider**

- **The obstacles will only be axis aligned rectangles.**
- **You can re-purpose the point inside the square code from Project1**
- **Using a line intersection algorithm might be helpful.**
- **Make sure that your collision checker accounts for all corner cases.**

(b) **Implement RTP for rigid body motion planning. At a minimum, your planner must derive from ompl::base::Planner and correctly implement the solve(), clear(), and getPlannerData() functions. Solve() should emit an exact solution path when one is found. If time expires, it**

should also emit an approximate path that ends at the closest state to the goal in the tree. It may be helpful to start from an existing planner, and modify it, such as RRT. You can check the source files of RRT.h, RRT.cpp, and the online documentation available here. Note that:

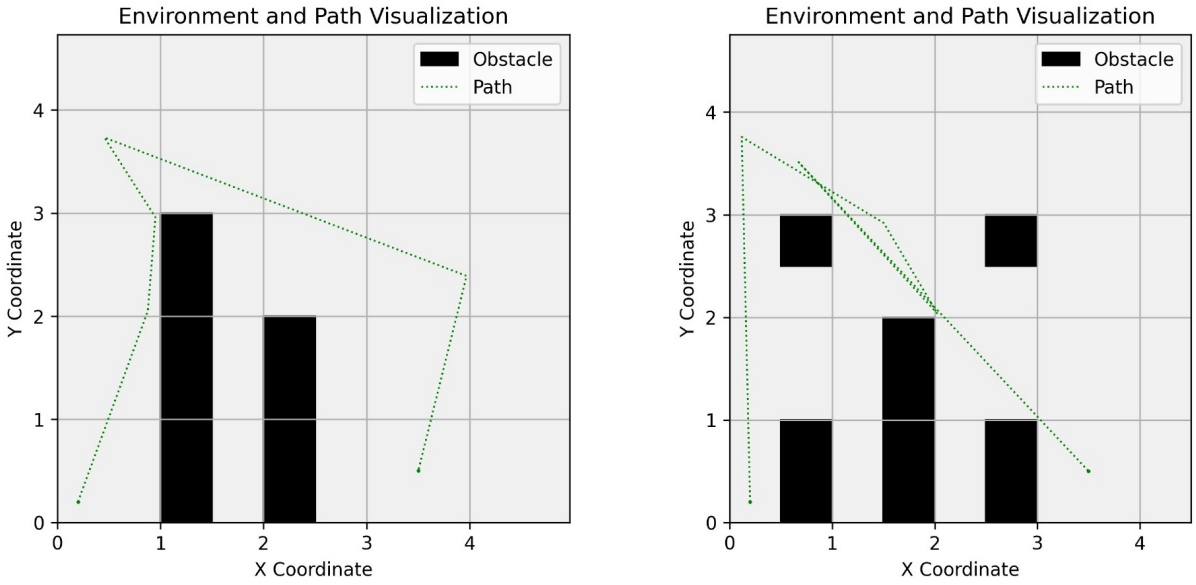
- **You need to fill the implementations in RTP.h and RTP.cpp**
- **Your planner does not need to know the geometry of the robot or the environment, or the exact C-space it is planning in. These concepts are abstracted away in OMPL so that planners can be implemented generically.**

(c) **Fill-out the missing functions at PlanningRTP.cpp You can check the OMPL demos on how to setup different planning problems here. The RigidBodyPlanning might be the most relevant.**

- **Develop at least two interesting environments for your robot to move in. Bounded environments with axis-aligned rectangular obstacles are sufficient.**
- **Perform motion planning in your developed environments for the point robot and the square robot. Collision checking must be exact, and the robot should not leave the bounds of your environment. Note, instead of manually constructing the state space for the square robot, OMPL provides a default implementation of the configuration space $R^2 \times S^1$, called ompl::base::SE2StateSpace**
- **Include in your report the images of your environments, corresponding solution paths, and start and goal queries. In your report you will need to provide 4 images. Two for the point robot, and defined environments and two for the square robot and defined environments.**
- **You can use the visualise.py script provided to you for visualizing the environments and the paths. You can run the script with: python3 visualise.py –obstacles obs file –path path file**

i. Point Robot

- Start Configuration for Point robot is $(x, y) : (0.2, 0.2)$
- Goal Configuration for Point robot is $(x, y) : (3.5, 0.5)$
- Environment bounds are $(lowBound, highBound) : (0.0, 4.0)$ in both the axes.



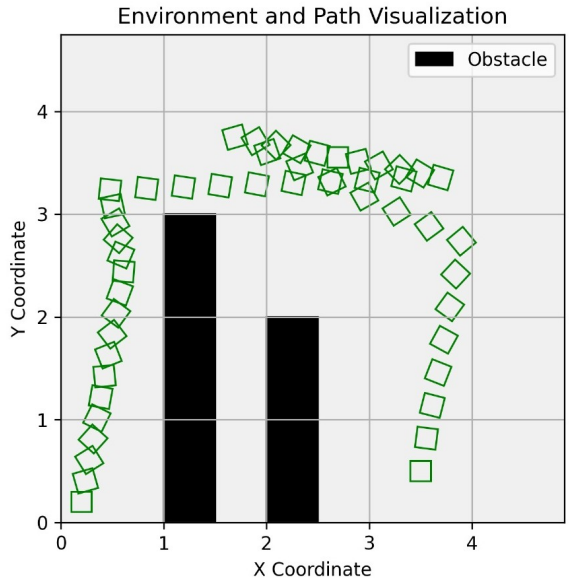
(a) Environment 1

(b) Environment 2

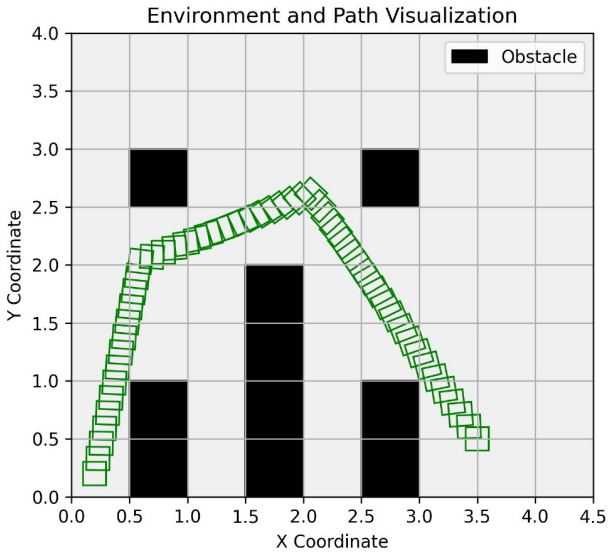
Figure 4: Results obtained for Point robot

ii. Square Robot

- Start Configuration for Square robot is (x, y, θ) : $(0.2, 0.2, 0)$
- Goal Configuration for Square robot is (x, y, θ) : $(3.5, 0.5, 0)$
- Environment bounds are $(lowBound, highBound)$: $(0.0, 4.0)$ in both the axes.



(a) Environment 1



(b) Environment 2

Figure 5: Results obtained for Square robot

(d) **Download and modify appropriately the KinematicChain environment to benchmark your planner. If all of the planners fail to find a solution, you will need to increase the computation time allowed. Benchmark with a kinematic chain of 20 and 10 links. Compare and contrast the solutions of your RTP with EST, and RRT planners. Elaborate on the performance of your planner RTP. Conclusions must be presented quantitatively from the benchmark data. Consider the following metrics: computation time, path length, and the number of states sampled (graph states). In your submitted files include the 2 generated databased named benchmarkChain20.db and benchmarkChain10.db**

The RTP is evaluated against RRT and EST on the following metrics:

- Computation time
- Path Length
- Number of States sampled (graph states)

These are analyzed using the Box Plots in PlannerArena.

The below box plots shows comparison between the planners for 10 Link Kinematic chain.

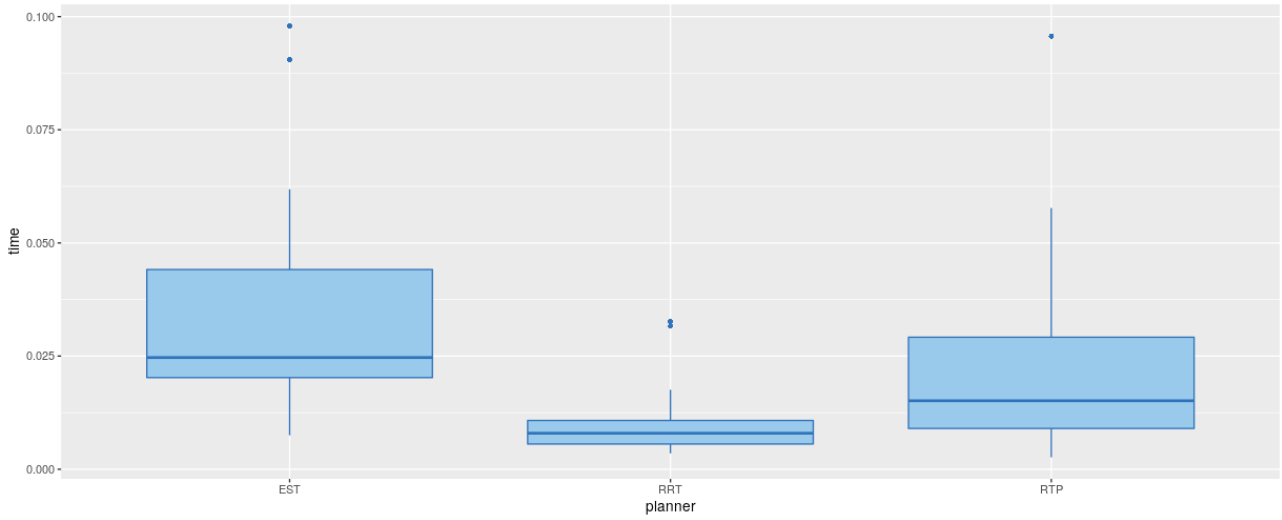


Figure 6: Computation Time

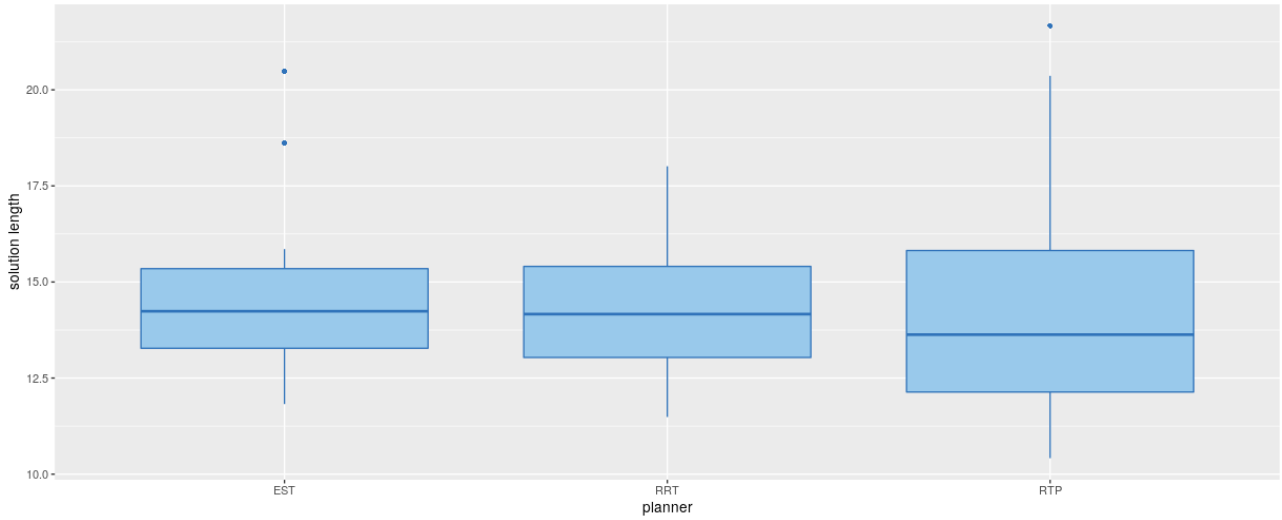


Figure 7: Path Length

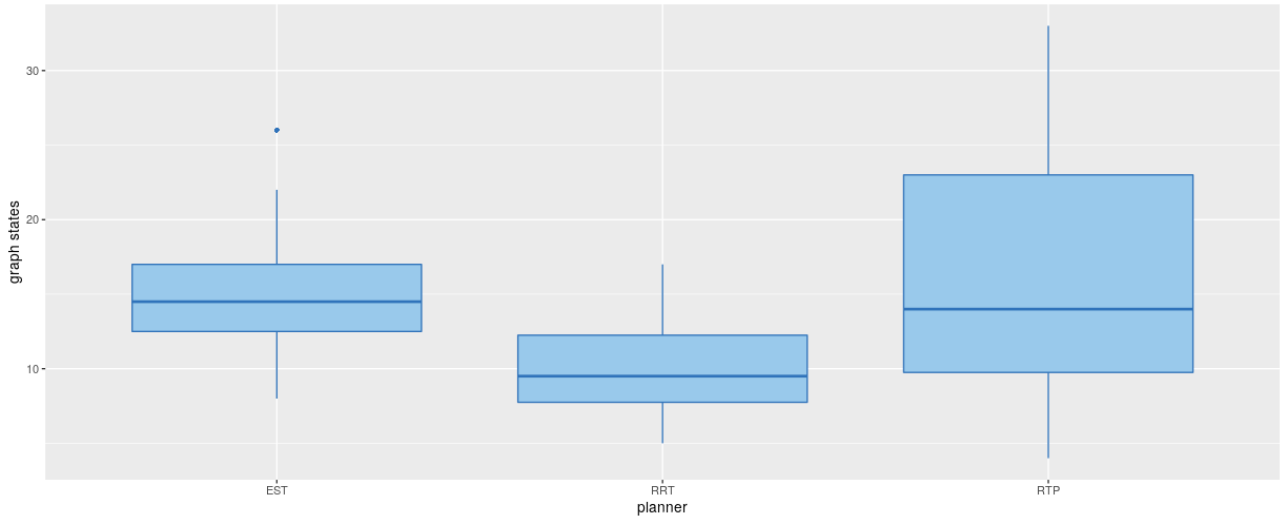


Figure 8: Number of States Sampled

The from above figures the Table 2 is generated which shows comparison between the planners for 10 Link Kinematic chain.

Computation time	**EST**	**RRT**	**RTP**
Min	Highest	Close to RTP	Least time
Max	Highest	Least	Significantly higher than RRT
Median	Highest	Least	Higher than RRT
1^{st} Quartile	Highest	Least	Higher than RRT
3^{rd} Quartile	Highest	Least	Higher than RRT
Inter-Quartile Range	Widest	Smallest	Close to EST
Path Length	**EST**	**RRT**	**RTP**
Min	Highest	Close to EST	Least
Max	Least	Between EST & RTP	Highest
Median	Similar to RRT	Higher than RTP	Least
1^{st} Quartile	Highest	Close to EST	Least
3^{rd} Quartile	Least	Between EST and RTP	Highest
Inter-Quartile Range	Lowest	Similar to EST	Highest
States Sampled	**EST**	**RRT**	**RTP**
Min	Highest	Between EST & RTP	Least
Max	Between RRT & RTP	Least	Highest
Median	Similar to RTP	Least	Higher than RRT
1^{st} Quartile	Highest	Least	Higher than RRT
3^{rd} Quartile	Between RRT & RTP	Least	Highest
Inter-Quartile Range	Similar to RRT	Small	Large

Table 2: 10 Link Kinematic Chain

Summarizing from the above Figures and Graphs for 10 link kinematic chain, we conclude that:

- The median computation time for RTP is lower than EST but slightly higher than RRT. However the time taken for coming to a solution for the RTP varies a lot for multiple run.
- The median path length for RTP is the least. However, it varies the largest from computation to computation.
- The median number of states sampled is quite higher than RRT and depends heavily on multiple runs of the planner.

The below box plots shows comparison between the planners for 20 Link Kinematic chain.

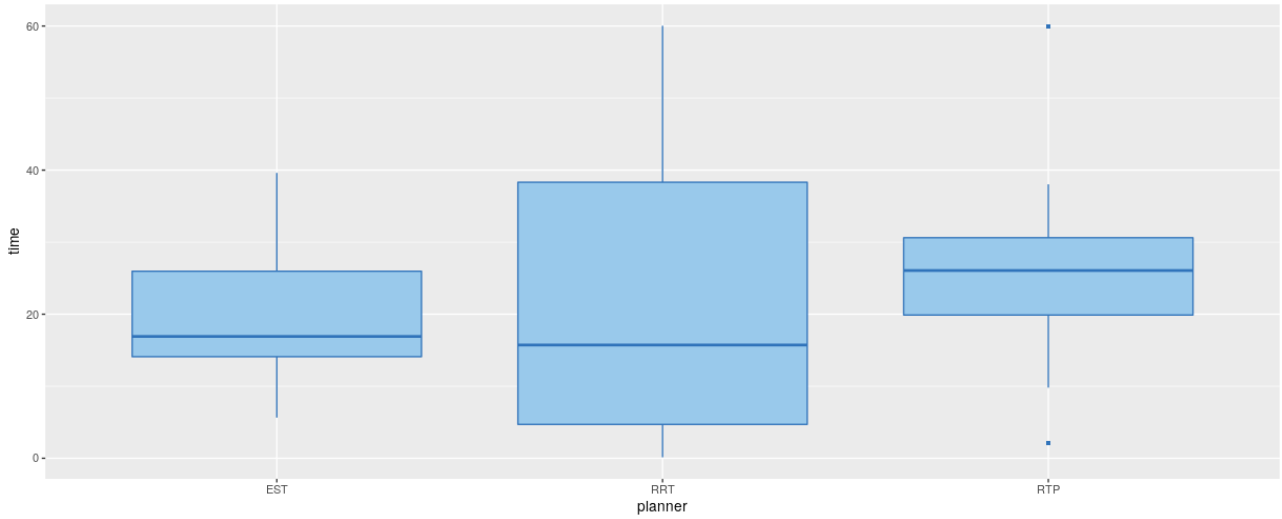


Figure 9: Computation Time

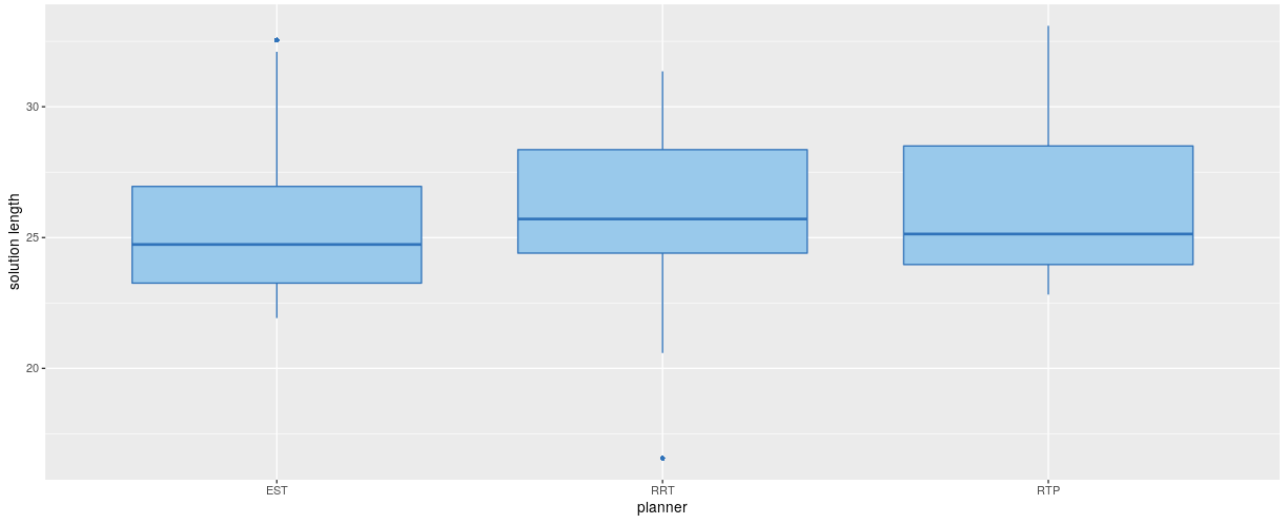


Figure 10: Path Length

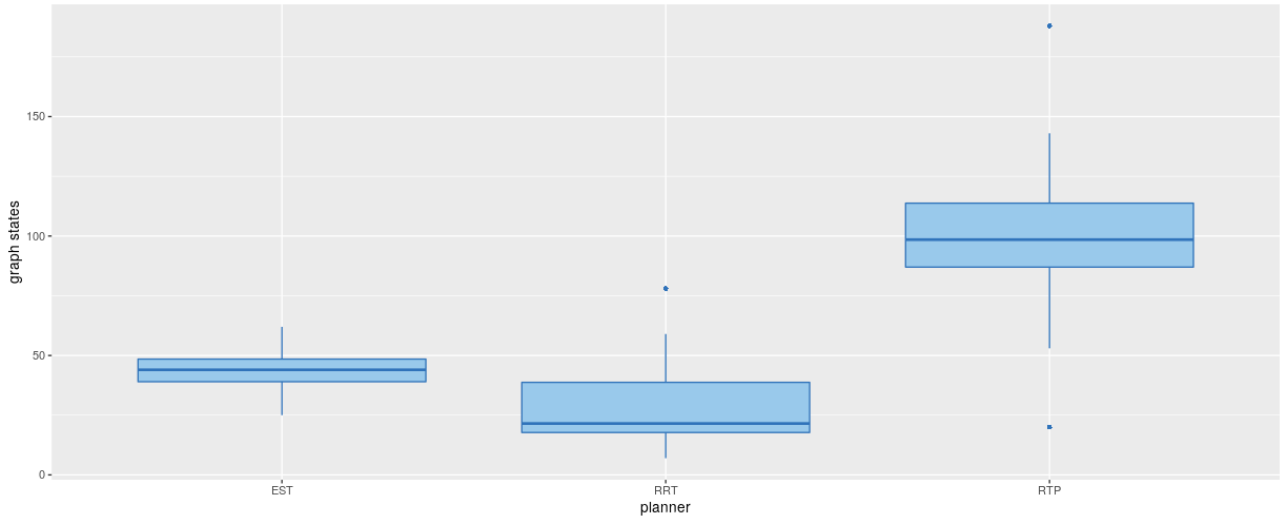


Figure 11: Number of States Sampled

The from above figures the Table 3 is generated which shows comparison between the planners for 10 Link Kinematic chain.

Computation time	**EST**	**RRT**	**RTP**
Min	Between RRT & RTP	Least	Highest
Max	Similar to RTP	Highest	Least
Median	Similar to RRT	Least	Highest
1^{st} Quartile	Between RRT & RTP	Least	Highest
3^{rd} Quartile	Least	Highest	Between EST & RRT
Inter-Quartile Range	Similar to RTP	Large	Small
Path Length	**EST**	**RRT**	**RTP**
Min	Between RRT & RTP	Least	Highest
Max	Between RRT & RTP	Least	Highest
Median	Least	Highest	Between EST & RRT
1^{st} Quartile	Least	Highest	Close to RRT
3^{rd} Quartile	Least	Close to RTP	Highest
Inter-Quartile Range	Lowest	Similar to EST	Highest
States Sampled	**EST**	**RRT**	**RTP**
Min	Between RRT & RTP	Least	Highest
Max	Close to RRT	Least	Highest
Median	Between RRT & RTP	Least	Highest
1^{st} Quartile	Between RRT & RTP	Least	Highest
3^{rd} Quartile	Between RRT & RTP	Least	Highest
Inter-Quartile Range	Very Small	Between EST & RTP	Largest

Table 3: 20 Link Kinematic Chain

Summarizing from the above Figures and Graphs for 20 link kinematic chain, we conclude that:

- The median computation time for RTP is highest. The time taken for the RTP does not vary much from run to run.
- The median path length for RTP lower than RRT but higher than EST. The Inter-Quartile range is also the maximum for RTP but not very high indicating that multiple runs does not affect the path length significantly.
- The median number of states sampled is very high for RTP and number of sampled states does not vary much from run to run.
- Since RTP samples a lot of states we can safely say the reason for higher computation time for RTP is due to the higher sampling of the states.