RBE550: Motion Planning

Project 2

Student 1: Sumukh, Porwal
Student 2: Krutika, Muley

Theoretical Questions

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

   Solution:

   Bug 1 is an "exhaustive" search algorithm where the robot follows the obstacle's boundary completely before resuming towards the goal, ensuring no part of the obstacle is missed. In contrast, Bug 2 is a "greedy" algorithm, as the robot only follows the obstacle boundary until it can head directly towards the goal, aiming for the shortest possible path around obstacles.
   Bug 1 tends to have more predictable performance because it guarantees the robot will reach the goal if it's reachable, even if that means taking a longer path around obstacles. Bug 2, while faster in many cases, can result in less predictable behavior, especially in complex environments. Bug 1 is safer and more reliable as it thoroughly explores, whereas Bug 2 sacrifices thoroughness for speed, which may lead to suboptimal paths.

2. A fundamental part of A^* search is the use of a heuristic function to avoid exploring unnecessary edges. To guarantee that A^* will find the optimal solution, the heuristic function must be admissible. 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 from node n to its adjacent node n'.

   Answer the following:

   (a) 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.

   (b) 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.

   Solution:

   (a) Define h_a as:

   $$h_a = \sqrt{(x - g_x)^2 + (y - g_y)^2}$$

   This is the Euclidean distance between the current cell (x, y) and the goal cell (g_x, g_y). Since the Euclidean distance is the straight-line distance between two points, and in grid-based movement, the actual path cannot be shorter than this straight-line distance, this heuristic never overestimates the true cost. Hence, it is admissible.

   Now, define h_b as:

   $$h_b = (x - g_x) + (y - g_y)$$

   This is the Manhattan distance between the current cell and the goal cell. While this is a reasonable approximation in environments where movement is restricted to horizontal or vertical directions, it underestimates the cost in environments that allow diagonal movement. Since it does not account for diagonal movement, it may underestimate the true cost and, in cases of complex movement, overestimate the cost. Therefore, h_b can be non-admissible.

   In summary, h_a is admissible as it represents a lower bound on the true cost, while h_b may not always reflect the actual path cost, thus being non-admissible.

(b) Let h_1 and h_2 be two consistent heuristics. Define the new heuristic as:

$$h(n) = \max(h_1(n), h_2(n))$$

To prove that h is consistent, we need to show that for all consecutive nodes n and n', the following holds:

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

By definition of h, we know that:

$$h(n) = \max(h_1(n), h_2(n)) \quad \text{and} \quad h(n') = \max(h_1(n'), h_2(n'))$$

Since h_1 and h_2 are consistent, we know that:

$$h_1(n) \leq T(n, n') + h_1(n') \quad \text{and} \quad h_2(n) \leq T(n, n') + h_2(n')$$

Therefore, we can write:

$$\max(h_1(n), h_2(n)) \leq T(n, n') + \max(h_1(n'), h_2(n'))$$

This shows that $h(n)$ is also consistent, as the maximum of two consistent heuristics cannot violate the consistency condition.
Hence, $h(n) = \max(h_1(n), h_2(n))$ is a consistent heuristic.

3. Consider workspace obstacles A and B. If $A \cap B \neq \emptyset$, do the configuration space obstacles Q_A and Q_B always overlap? If $A \cap B = \emptyset$, is it possible for the configuration space obstacles Q_A and Q_B to overlap? Justify your claims for each question.

Solution:

(a) If $A \cap B \neq \emptyset$, then the workspace obstacles A and B overlap in the physical space, meaning that there is some region that is common between them. In the configuration space, which accounts for both the position and orientation of the robot, the obstacles Q_A and Q_B will also overlap. This is because any configuration that places the robot in the overlapping region in the workspace will be considered as part of both Q_A and Q_B. As a result, $Q_A \cap Q_B \neq \emptyset$ — there will always be some configurations that result in a collision with both A and B, so the configuration space obstacles will overlap.

(b) If $A \cap B = \emptyset$, meaning that the obstacles A and B do not intersect in the workspace, it is still possible for the configuration space obstacles Q_A and Q_B to overlap. This can occur because the configuration space takes into account the size and shape of the robot as well as its orientation. Even if A and B are not physically overlapping in the workspace, the robot may still occupy parts of both A and B simultaneously in certain configurations, leading to an overlap in the configuration space. For example, if the robot is large or has a particular shape that extends into both obstacle regions when placed near both A and B, the configuration space obstacles can overlap, even though the workspace obstacles do not. Therefore, $Q_A \cap Q_B \neq \emptyset$ is possible even when $A \cap B = \emptyset$.

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.

Answer the following:

(a) 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.

(b) 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.

Solution:

(a) To construct the visibility graph, we need to check whether every pair of vertices (including the start and goal points) can be connected by an edge without intersecting any obstacles.

- Step 1: For each pair of vertices, check if the straight line segment between them intersects any obstacle.
- Step 2: If no intersection is found, add an edge between these vertices.

Given that there are n vertices (including the vertices of the polygonal obstacles, the start point, and the goal point), we need to perform this check for every pair of vertices. The total number of vertex pairs is $O(n^2)$ because we are connecting each vertex to every other vertex.

For each pair, we need to check whether the line segment connecting them intersects any obstacle. Assuming there are $O(n)$ edges in total across all obstacles, the intersection check for each pair of vertices can be done in $O(n)$ time.

Thus, the overall time complexity to construct the visibility graph is $O(n^2)$ for checking all vertex pairs and $O(n)$ for each intersection check. Therefore, the time complexity is:

$$O(n^2) \times O(n) = O(n^3)$$

Pseudocode for constructing the visibility graph:

Algorithm 1 Construct Visibility Graph

```
function ConstructVisibilityGraph($vertices$, $obstacles$)
    $edges \leftarrow \emptyset$                                        ▷ Initialize the set of edges
    for $v_i$ in $vertices$ do
        for $v_j$ in $vertices$ where $i \neq j$ do
            $intersects \leftarrow False$
            for $obstacle$ in $obstacles$ do
                if line segment $(v_i, v_j)$ intersects $obstacle$ then
                    $intersects \leftarrow True$
                    break                    ▷ No need to check further if there is an intersection
            if $intersects = False$ then
                add edge $(v_i, v_j)$ to $edges$
    return $edges$                                ▷ Return the constructed visibility graph
```

This algorithm checks each pair of vertices and adds an edge if the line segment does not intersect any obstacle.

(b) Yes, you can use the visibility graph to plan a path from the start to the goal. The visibility graph essentially represents all possible straight-line paths between the vertices of the obstacles, as well as the start and goal. Once the visibility graph is constructed, path planning becomes a graph search problem where you seek the shortest path from the start vertex to the goal vertex.

To find the shortest path, you can use Dijkstra's algorithm or the A^* algorithm, both of which are well-suited for finding the shortest path in a graph where edge weights represent distances.

The run-time complexity of Dijkstra's algorithm is $O((n+m)\log n)$, where n is the number of vertices and m is the number of edges in the visibility graph. The same complexity applies to A^* as long as the heuristic function used is admissible and consistent.

Thus, the upper bound for the run-time of the path-planning algorithm is:

$$O((n+m)\log n)$$

Where:

- n is the number of vertices in the visibility graph (including obstacle vertices, start, and goal).
- m is the number of edges in the visibility graph (which is at most $O(n^2)$, but could be smaller depending on the obstacle layout).

In conclusion, after constructing the visibility graph, you can use a graph search algorithm to plan the optimal path from the start to the goal efficiently.

Programming Component

1. 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 squre code from Project1.
   - Using a line intersection algorithm might be helpful.
   - Make sure that your collision checker accounts for all corner cases

2. 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 $\mathcal{C}$-space it is planning in. These concepts are abstracted away in `OMPL` so that planners can be implemented generically.

3. 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 $\mathbb{R}^2 \times \mathbb{S}^1$, 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>`
     We have provided you with example path.txt and example obstacles.txt as reference file for the visualizer. You should make your own files and provide them under the build directory.

Solution:

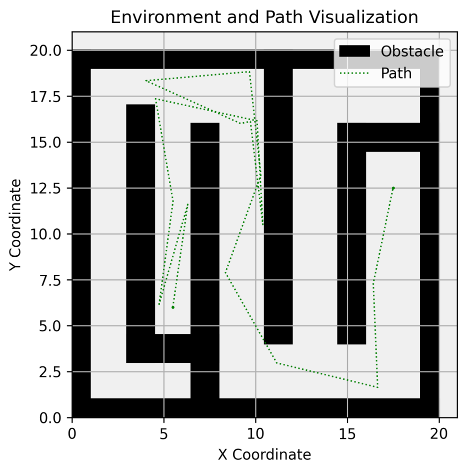

Figure 1: Point Robot Path in Maze Environment

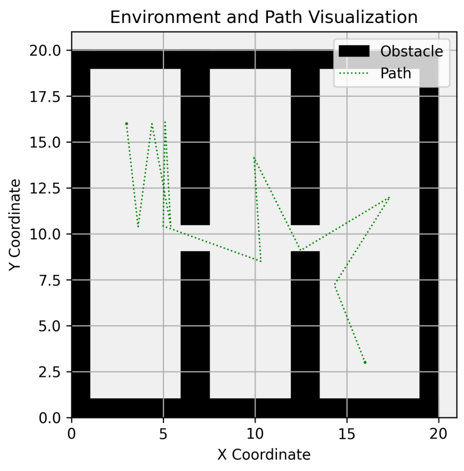

Figure 2: Point Robot Path in Flappy Bird Environment

Start for the Point Robot in Figure 1 is (5.5, 6.0) and the end goal state is (17.5, 12.5) in $\mathbb{R}^2$ space.
Similarly, Start for the Point Robot in Figure 2 is (16.0, 3.0) and the end goal state is (3.0, 16.0) in $\mathbb{R}^2$ space.

Both Figures (1) and (2) illustrate that the point robot follows a stochastic path to the goal, where 98% of the time, it selects the next state randomly and verifies the feasibility of motion to that point. The robot chooses the goal state as the next state with a 2% probability. Evidently, the path is non-optimal, being randomly generated, and includes oscillatory or back-and-forth movements.

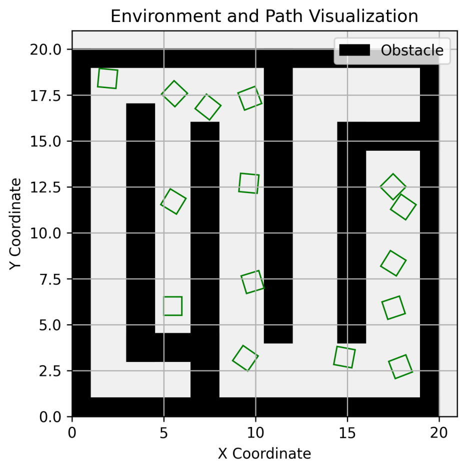

Figure 3: Box Robot Path in Maze Environment

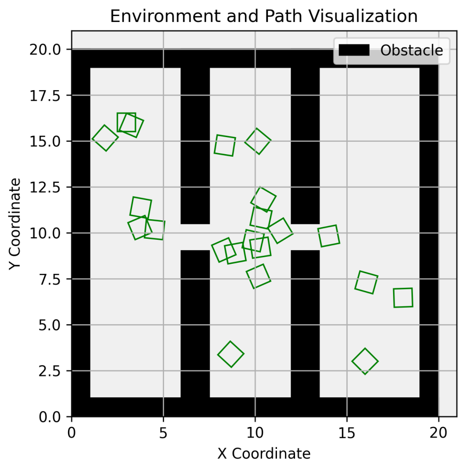

Figure 4: Box Robot Path in Flappy Bird Environment

Start for the Box Robot in Figure 3 is (5.5, 6.0) and $yaw = 0$ and the end goal state is (17.5, 12.5) and $yaw = \frac{\pi}{4}$ in $\mathbb{R}^2 \times \mathbb{S}^1$ space.
Similarly, Start for the Box Robot in Figure 4 is (16.0, 3.0) & $yaw = 0$ and the end goal state is (3.0, 16.0) & $yaw = \frac{\pi}{4}$ in $\mathbb{R}^2 \times \mathbb{S}^1$ space.

Figures (3) and (4) demonstrate that the box robot navigates towards the goal by choosing its next state randomly 98% of the time, checking whether the movement to that state is possible. With a 2% probability, it directly selects the goal state. As the path is randomly generated, it is clearly non-optimal, often resulting in movements in incorrect directions and repetitive back-and-forth actions.

4. 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`

   Solution:
   Each planner was executed **20** times for at most **120** seconds. Memory is limited at **1024MB**.

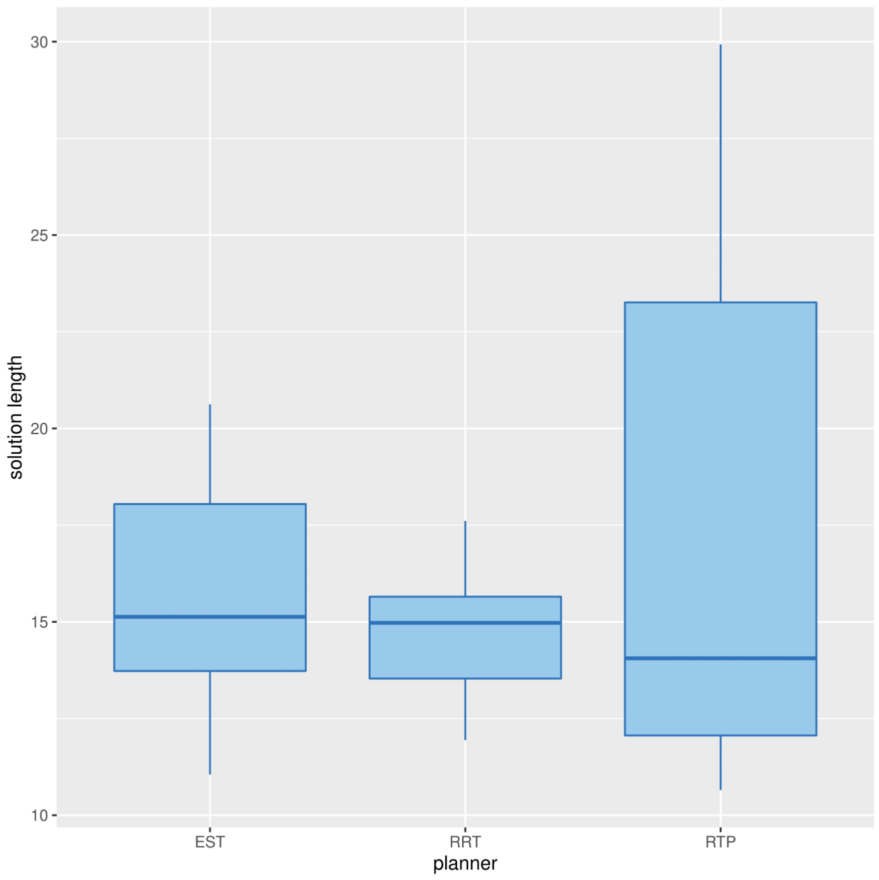

Figure 5: Path Length Comparison between EST, RRT and RTP with **10** links

Figure 5 insights:

- EST (Expansive Space Trees):
  The median solution length for EST is approximately 15, and the range of solution lengths lies between 12 and 21. There is moderate variability in solution lengths, indicating that EST produces relatively consistent results in terms of path length. The slightly larger spread suggests that EST explores the space thoroughly, but this can sometimes result in longer paths, depending on the sampled states.
- RRT (Rapidly-exploring Random Trees):
  RRT has the lowest median solution length, which is slightly less than 15. This suggests that RRT tends to find more direct paths to the goal. The range of solution lengths is also quite small compared to the other planners, indicating that RRT consistently finds shorter paths. This performance aligns with RRT's greedy nature in expanding towards unexplored areas, allowing it to find shorter, more efficient paths.
- RTP (Random Tree Planning):
  RTP exhibits the widest range of solution lengths, with a median close to 13 and values ranging from 10 to nearly 30. The large spread indicates that RTP's performance is highly variable. In some cases, it finds shorter paths (around 10), while in others, it finds significantly longer paths. RTP's random exploration strategy without heavy direction bias means that its solution lengths can fluctuate greatly. Sometimes it finds direct paths, but other times it explores unnecessarily, leading to longer solutions.

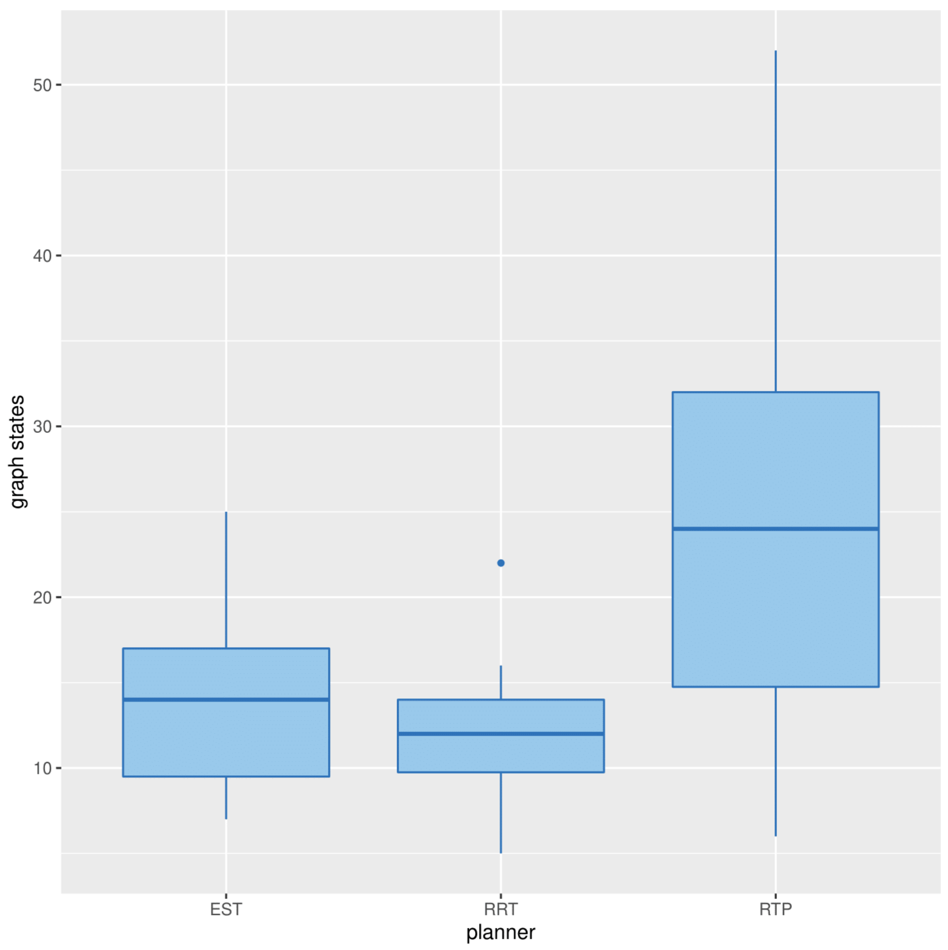

Figure 6: Number of States Comparison between EST, RRT and RTP with **10** links

Figure 6 insights:

- EST (Expansive Space Trees):
  The boxplot shows that the median number of graph states generated is around 13-15.The range is moderate, with some lower values just above 6 and some higher values close to 25. EST's approach involves sampling nodes in regions where coverage is less, which results in a balanced exploration of space, hence the moderate range of states.
- RRT (Rapidly-exploring Random Trees):
  The median is slightly lower compared to EST, closer to 12-13. The overall range is smaller, indicating that RRT generates fewer states with less variance. RRT's greedy exploration method emphasizes reaching unexplored areas, which may lead to fewer states compared to EST. The smaller range suggests a more consistent performance, as RRT systematically grows toward unexplored areas.
- RTP (Random Tree Planning):
  RTP has the widest range of graph states, with the median graph state count much higher, around 24-25, and outliers that extend beyond 50. This high variability is expected since RTP randomly selects the next state, occasionally favoring the goal. This randomness introduces unpredictability in the number of states generated. The larger number of graph states suggests RTP explores more potential paths, including many redundant or less optimal ones. RTP's tendency to occasionally move toward the goal with small probability makes it less structured compared to the directed strategies of EST and RRT, which is why RTP appears more inefficient (larger number of graph states).

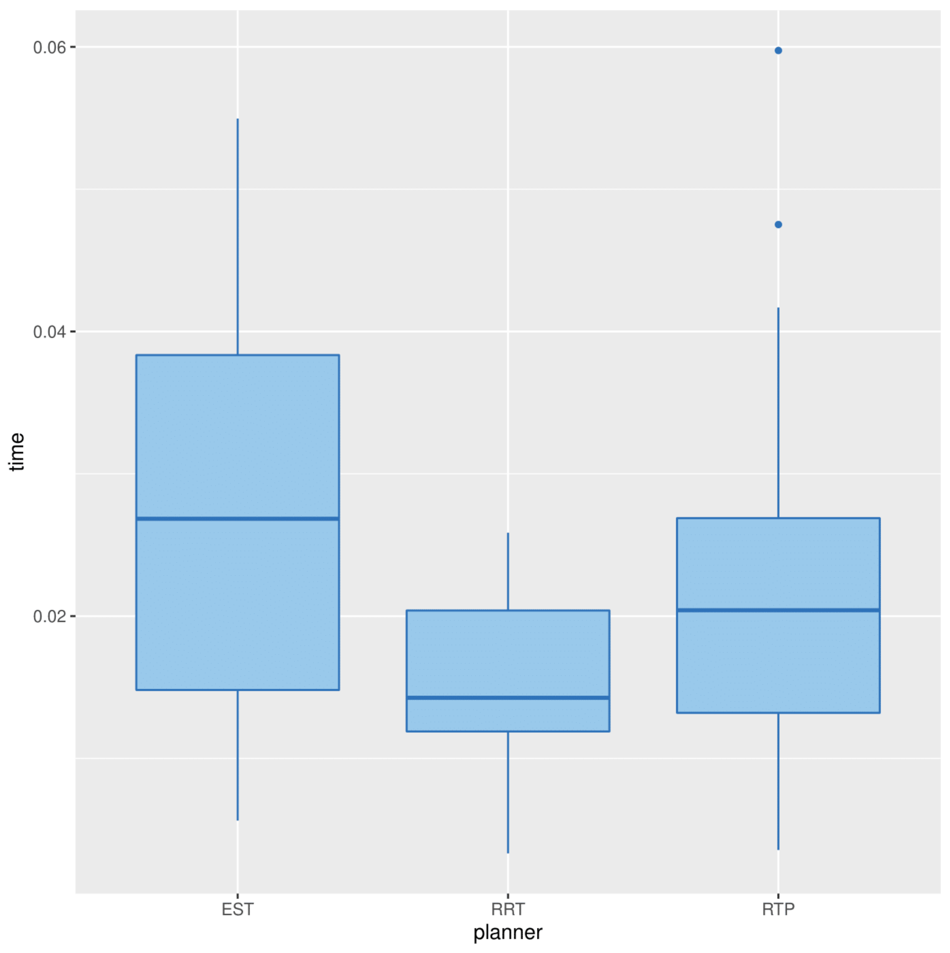

Figure 7: Computation Time Comparison between EST, RRT and RTP with **10** links

Figure 7 insights:

- EST (Expansive Space Trees):
  The median time taken by EST is around 0.024, and it shows a relatively large spread of values, ranging from just under 0.005 to above 0.055. The variability is higher compared to RRT, meaning that EST can sometimes take significantly longer depending on the specific scenario. EST's exploration strategy, which focuses on sampling areas that have low coverage, can result in time variability based on how evenly the space is sampled.
- RRT (Rapidly-exploring Random Trees):
  RRT has the lowest median time, indicating it is the fastest among the three planners. The median is close to 0.012, and the range is small, making it a more consistent performer in terms of time. RRT's greedy nature in rapidly expanding into unexplored areas allows it to find a path faster, which is reflected in the lower median time and smaller variability. This suggests that RRT is efficient in finding a solution with fewer time fluctuations.
- RTP (Random Tree Planning):
  The median time for RTP is higher than that of RRT, around 0.02, and it shows a larger range, with a few outliers taking time close to 0.06. RTP's wide range and higher median time indicate that its random exploration strategy, which occasionally favors goal direction, is more time-consuming than the directed strategies of RRT and even EST. The random nature of state selection can lead RTP to explore unnecessary areas, hence the higher overall time compared to the others.

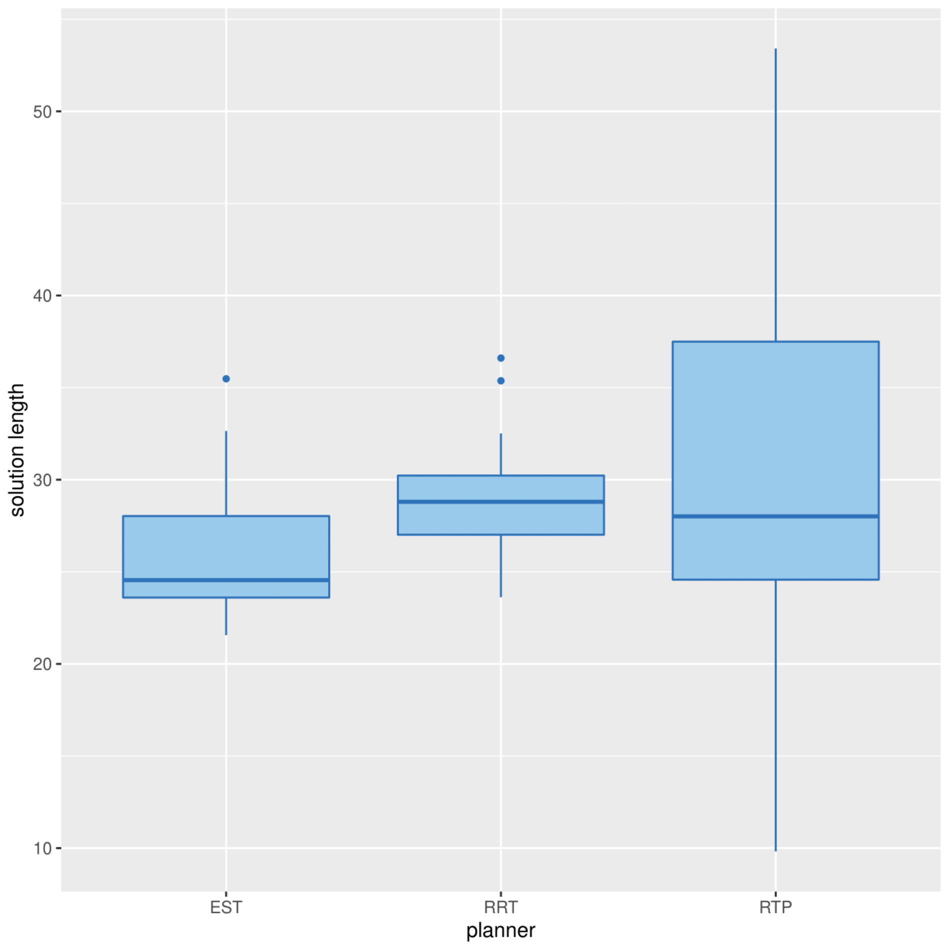

Figure 8: Path Length Comparison between EST, RRT and RTP with **20** links

Figure 8 insights:

- EST (Expansive Space Trees):
  The median solution length is lowest for EST is approximately 25, and the range of solution lengths lies between 22 and 33. There is moderate variability in solution lengths, indicating that EST produces relatively consistent results in terms of path length. The slightly larger spread suggests that EST explores the space thoroughly, but this can sometimes result in longer paths, depending on the sampled states.
- RRT (Rapidly-exploring Random Trees):
  RRT has the median solution length slightly less than 29. This suggests that EST tends to perform better with more number of links than RRT. The range of solution lengths is quite small compared to the other planners, indicating that RRT is consistent with finding paths. This performance aligns with RRT's greedy nature in expanding towards unexplored areas, allowing it to find shorter, more efficient paths.
- RTP (Random Tree Planning):
  RTP exhibits the widest range of solution lengths, with a median close to 28 and values ranging from 10 to nearly 53. The large spread indicates that RTP's performance is highly variable. In some cases, it finds shorter paths (around 10), while in others, it finds significantly longer paths. RTP's random exploration strategy without heavy direction bias means that its solution lengths can fluctuate greatly. Sometimes it finds direct paths, but other times it explores unnecessarily, leading to longer solutions.

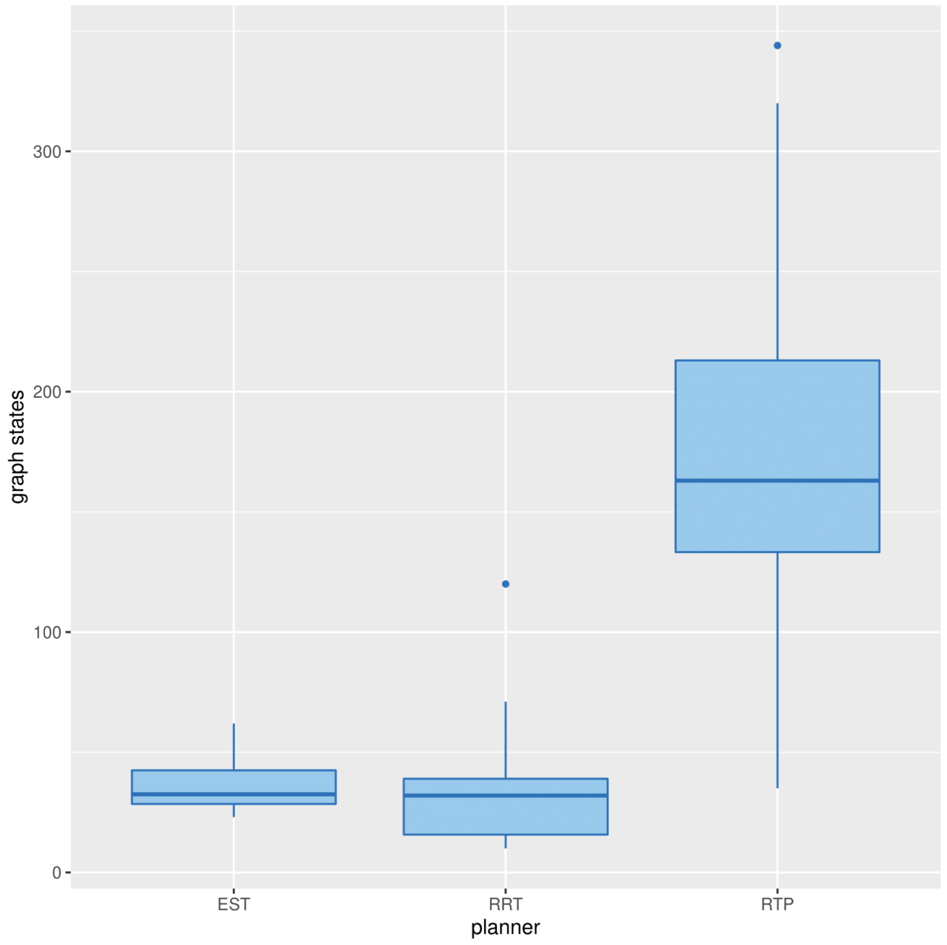

Figure 9: Number of States Comparison between EST, RRT and RTP with **20** links

Figure 9 insights:

- EST (Expansive Space Trees):
  The boxplot shows that the median number of graph states generated is around 35.The range is smallest, with some lower values just above 25 and some higher values close to 60. EST's approach involves sampling nodes in regions where coverage is less, which results in a balanced exploration of space, hence the moderate range of states.
- RRT (Rapidly-exploring Random Trees):
  The median is slightly lower compared to EST, closer to 23. The overall range is more than taht of EST, indicating that EST generates fewer states with less variance than RRT in case with more number of links. RRT's greedy exploration method emphasizes reaching unexplored areas, which may lead to few more states compared to EST. Their is also one outlier around 120.
- RTP (Random Tree Planning):
  RTP has the widest range of graph states, with the median graph state count much higher, around 165, and outliers that extend beyond 340 and range varying from 40 to 320. This high variability is expected since RTP randomly selects the next state, occasionally favoring the goal. This randomness introduces unpredictability in the number of states generated. The larger number of graph states suggests RTP explores more potential paths, including many redundant or less optimal ones. RTP's tendency to occasionally move toward the goal with small probability makes it less structured compared to the directed strategies of EST and RRT, which is why RTP appears more inefficient (larger number of graph states).

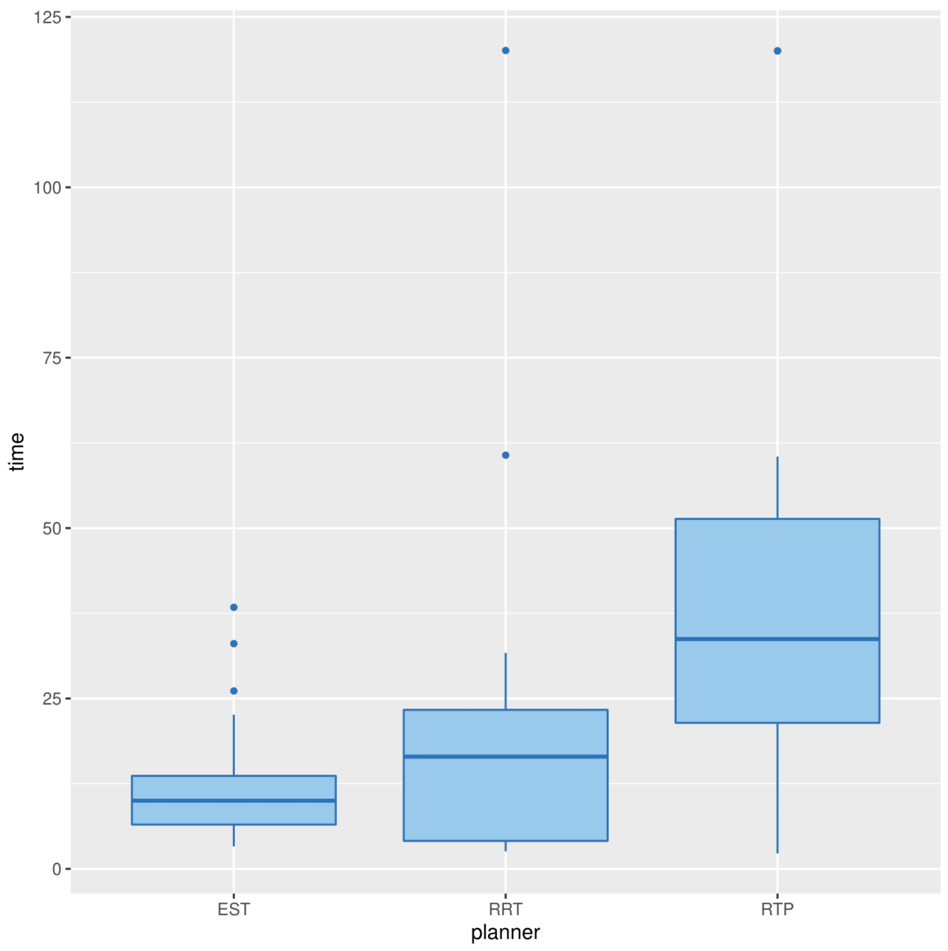

Figure 10: Computation Time Comparison between EST, RRT and RTP with **20** links

Figure 10 insights:

- EST (Expansive Space Trees):
  The median time taken by EST is around 10, and it shows a relatively small spread of values, ranging from just under 4 to above 23. The variability is lesser compared to RRT, meaning that EST can sometimes be significantly shorter depending on the specific scenario. EST's exploration strategy, which focuses on sampling areas that have low coverage, can result in time variability based on how evenly the space is sampled.
- RRT (Rapidly-exploring Random Trees):
  RRT has higher median time than EST. The median is close to 15, and the spread is also bigger than EST, ranging from 3.5 to 31. RRT's greedy nature in rapidly expanding into unexplored areas can sometimes lead to more than than EST there are outliers even as high as around 120.
- RTP (Random Tree Planning):
  The median time for RTP is higher than that of RRT, around 30, and it shows a larger range from 3 to 60, with a few outliers taking time close to 120. RTP's wide range and higher median time indicate that its random exploration strategy, which occasionally favors goal direction, is more time-consuming than the directed strategies of RRT and even EST. The random nature of state selection can lead RTP to explore unnecessary areas, hence the higher overall time compared to the others.