RBE550: Motion Planning

Project 3

Student 1: Sumukh, Porwal
Student 2: Piyush, Thapar

Theoretical Questions

1. **Recall the visibility graph method. Similar to the PRM, the visibility graph also captures the continuous space using a graph structure. Compare these two methods. For each method, provide at least one scenario in which it would work well while the other would not. Justify your answer.**

   Solution:
   Visibility Graph:
   The visibility graph method is a deterministic graph-based approach where nodes represent the start, goal, and vertices of obstacles in the environment. Edges are added between nodes if there is a direct line-of-sight between them without intersecting obstacles. The method then uses graph search algorithms, such as Dijkstra's or A*, to find the shortest path from the start to the goal.

   PRM:
   The Probabilistic Roadmap (PRM) method is a probabilistic sampling-based approach used for motion planning in high-dimensional spaces. The method consists of two phases:
   The learning phase involves randomly sampling points in the free space and connecting them to nearby points if the connection is collision-free.
   The query phase involves searching the graph using an algorithm like Dijkstra's or A* to find the shortest path between the start and goal points.

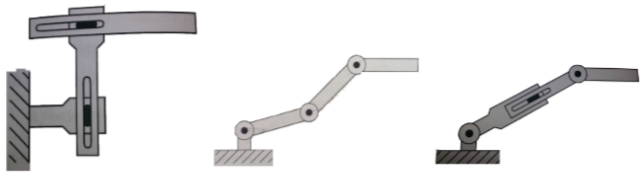
   Comparison:

   Visibility Graph Strengths:
   The visibility graph method works well in environments with polygonal obstacles in 2D space, where it can efficiently compute optimal paths by finding straight-line paths between obstacle vertices. The paths obtained are often optimal in terms of distance because of the direct line-of-sight edges.
   Scenario: Consider a 2D environment with a small number of convex polygonal obstacles. In this scenario, the visibility graph can efficiently capture all important edges of the free space and compute an optimal solution.
   PRM Weakness: In contrast, PRM may struggle in this case since randomly sampled nodes might fail to capture the structure of the free space as effectively as the deterministic obstacle vertices in the visibility graph, potentially leading to suboptimal or no solutions in narrow corridors.

   PRM Strengths:
   PRM is advantageous in high-dimensional spaces where the visibility graph becomes impractical due to the exponential growth in complexity with increasing dimensions. PRM scales better because it samples randomly and focuses on connecting feasible nodes, which reduces the number of connections needed in high-dimensional spaces.
   Scenario: Consider a robot navigating in a cluttered 6D configuration space (e.g., a robotic arm with six degrees of freedom). In this scenario, PRM would be able to sample the space more efficiently and produce feasible paths, while the visibility graph would be intractable due to the high-dimensional obstacle configuration.
   Visibility Graph Weakness: The visibility graph struggles in high-dimensional spaces because it requires checking the visibility between every pair of nodes, which is computationally expensive as the number of nodes grows exponentially with dimensionality.

   Conclusion:
   The visibility graph method excels in low-dimensional, structured environments with polygonal obstacles, where it can provide optimal paths. On the other hand, PRM is more suitable for high-dimensional spaces, where random sampling can efficiently capture the structure of the free space, and it can handle complex environments with a large number of degrees of freedom.

2. **For each of the three manipulators shown in Figure 1, determine the topology and dimension of the manipulator's configuration space.**



Solution:

(a) $\mathbb{R}^2$, dimensions = 2
(b) $S^1 \times S^1 \times S^1 = T^3$ (3-dimensional torus), dimensions = 3
(c) $S^1 \times \mathbb{R}^1 \times S^1$, dimensions = 3

3. **Answer the following questions about asymptotically optimal planners:**

   (a) **What is the core idea behind RRT*? That is, explain what modifications are done to RRT in order to make it asymptotically optimal. What methods are changed, and what changes are they?**

   Solution:

   The core idea behind RRT* (Rapidly-exploring Random Tree Star) is to modify the original RRT algorithm in such a way that it becomes asymptotically optimal. This means that, as the number of samples increases, the solution produced by RRT* converges to the optimal path.
   To achieve this, several modifications are made to the basic RRT algorithm:
   1. Rewiring of the Tree
   In RRT*, after a new node is added to the tree, the algorithm not only connects this node to the nearest node in the tree (like RRT), but it also attempts to rewire the tree. This means that it checks whether connecting existing nodes to this new node would result in a shorter path to the goal. Specifically, all nearby nodes within a certain radius are considered for rewiring, and if a shorter path can be found through the new node, the tree is updated accordingly.
   Change: This rewiring step improves the tree iteratively, optimizing the path over time, unlike RRT where once a node is connected, no further optimization is done.

   2. Choosing the Parent Node
   When adding a new node to the tree in RRT*, instead of simply connecting it to the nearest node (as in RRT), the algorithm selects the best parent from among all nodes in the neighborhood of the new node. The best parent is the one that results in the lowest cost to the new node, considering the cumulative cost from the start.
   Change: RRT* selects the parent that minimizes the cost-to-come (the cost from the start node to the new node) as opposed to RRT's nearest neighbor strategy, which only minimizes the distance.

   3. Radius for Nearby Nodes
   In RRT*, the search for nearby nodes to rewire the tree and to choose the best parent is conducted within a radius that shrinks as more nodes are added to the tree. This radius is inversely proportional to the number of nodes, ensuring that the rewiring step remains efficient as the number of samples increases.
   Change: The use of a shrinking radius ensures that the algorithm remains computationally feasible while still considering a sufficiently large neighborhood for rewiring early in the process.

   4. Cost Optimization
   RRT* explicitly optimizes the path cost. After adding a new node, it updates the costs of all the nodes in the vicinity that could potentially benefit from connecting through the new node, ensuring that the tree grows toward the optimal path.
   Change: RRT* focuses on improving the cost of the path iteratively, unlike RRT, which is focused on exploring the space without optimizing the solution.

   Summary of Changes:

- Rewiring: RRT* rewires the tree after adding a new node to improve path costs.
- Best Parent Selection: RRT* selects the parent that minimizes the cost-to-come.
- Shrinking Radius: RRT* uses a shrinking radius to keep the algorithm efficient over time.
- Cost Optimization: RRT* aims for path cost optimization, not just exploration.

These changes make RRT* asymptotically optimal because it guarantees that, as the number of samples increases, the solution approaches the optimal path. In contrast, RRT focuses on rapidly exploring the space and often produces a feasible but suboptimal path.

(b) **How does Informed RRT* improve upon RRT*. Is Informed RRT* strictly better than RRT* ? e.g., are there any cases where RRT* can find a better solution given the same sequence of samples?**

Solution:

Informed RRT* is an extension of RRT* that improves its efficiency, particularly in finding optimal paths. It introduces two key enhancements: focusing sampling to relevant areas of the space and refining the search based on the current best solution. Let's break down how Informed RRT* works, its improvements, and whether it's strictly better than RRT*.

Improvements of Informed RRT* Over RRT*:

Focusing Sampling on a Relevant Subspace:
In standard RRT*, nodes are sampled uniformly over the entire free space, regardless of how far they are from the current best solution. Informed RRT* improves this by focusing the sampling within an ellipsoidal region that is centered on the start and goal points and whose size depends on the cost of the current best path.
The ellipsoid encompasses all points that could potentially lead to a better path than the current best path. As the path improves, this ellipsoid shrinks, thus narrowing the search to more promising areas of the space.
Key Change: By restricting the search to this informed subset of the space, Informed RRT* reduces unnecessary exploration and focuses more on improving the existing solution.
Pruning Irrelevant Nodes:
Since Informed RRT* concentrates on improving the current solution, nodes and branches of the tree that cannot lead to a better solution are effectively pruned out. This allows the algorithm to focus its computational resources on the most relevant areas, further speeding up convergence.
Key Change: By disregarding regions of the space that cannot improve the current best path, Informed RRT* becomes more computationally efficient.
Maintaining Asymptotic Optimality:
Informed RRT* retains the key feature of RRT* — asymptotic optimality. It guarantees that as the number of samples increases, the solution will converge to the optimal path, just like RRT*. However, by focusing sampling within a smaller subspace, Informed RRT* tends to reach the optimal solution faster than RRT*.

While Informed RRT* is typically more efficient than RRT*, it is not strictly better in all cases. Here's why:

Informed Sampling Can Be Restrictive:
In some environments, the optimal path may lie outside of the ellipsoidal subspace that Informed RRT* initially restricts sampling to. For example, if there is a narrow passage or complex obstacle configuration that requires a circuitous route far from the direct path between start and goal, the ellipsoid may exclude the regions of space where the true optimal path lies. In such cases, RRT* can explore these regions and potentially find a better solution, whereas Informed RRT* might miss them early on due to its focus on the ellipsoid.
Identical Sequence of Samples:
Given the same sequence of random samples, RRT* and Informed RRT* will not behave the same. RRT* can explore more of the space because it samples uniformly, while Informed RRT* focuses only on the region that seems relevant at a given time. If the optimal solution lies in a region outside of the initial ellipsoid, RRT* might discover it earlier, while Informed RRT* would not due to its restricted sampling.
Convergence Speed vs. Completeness:
Informed RRT* is designed to converge faster to an optimal solution, but this focus on efficiency might come at the cost of completeness in very complex environments. While RRT* explores more uniformly, it may eventually find a better solution in a challenging environment, though potentially at the cost of slower convergence.

Summary:
Informed RRT* improves upon RRT* by focusing the sampling in areas that are more likely to lead to better solutions, reducing unnecessary exploration and speeding up convergence to the optimal path.
Informed RRT* is not strictly better than RRT* in all cases. In environments where the optimal path lies outside the ellipsoidal subspace, RRT could potentially find a better solution* given the same sequence of samples, as it explores the space uniformly and does not restrict sampling early on.
Thus, while Informed RRT* is often more efficient, RRT may outperform it in certain complex environments* where the optimal path is non-obvious and lies outside of the ellipsoid region.

4. **Suppose five polyhedral bodies float freely in a 3D world. They are each capable of rotating and translating. If these are treated as "one" composite robot, what is the topology of the resulting configuration space (assume that the bodies are not attached to each other)? What is the dimension of the composite configuration space?**

Solution:

When dealing with multiple polyhedral bodies that can rotate and translate independently in a 3D world, the configuration space (C-space) of the composite robot can be understood by analyzing the degrees of freedom (DOFs) for each individual body and then generalizing this to the entire system.

Configuration Space for One Polyhedral Body:

Each polyhedral body floating in 3D space has two types of motion:
Translation:
A polyhedron can translate freely in three-dimensional space, giving it 3 degrees of freedom (DOFs) corresponding to movement along the x, y and z axes.
Rotation:
A polyhedron can also rotate around three independent axes, which corresponds to 3 rotational DOFs (roll, pitch, and yaw).
Thus, the configuration space C_i for a single polyhedral body is the product of a 3D translation space and a 3D rotation space:

$$C_i = \mathbb{R}^3 \times SO(3)$$

where:
$\mathbb{R}^3$ represents the space of translations (free movement in 3D),
$SO(3)$ represents the space of rotations (the group of rotations in 3D).

The dimension of this configuration space is:

$$dim(C_i) = 3(tanslation) + 3(rotation) = 6$$

Composite Configuration Space for Five Polyhedra:
If we treat all five polyhedral bodies as one composite robot, each body contributes 6 DOFs independently. Therefore, the total configuration space $C_{composite}$ for the entire system is the Cartesian product of the configuration spaces of each body:

$$C_{composite} = (\mathbb{R} \times SO(3))^5$$

This means the configuration space is the product of five independent 6-dimensional configuration spaces, one for each body.

Dimension of the Composite Configuration Space:
Since each polyhedral body contributes 6 degrees of freedom, and there are 5 polyhedral bodies, the total dimension of the configuration space is:

$$dim(C_{composite}) = 6 \times 5 = 30$$

Programming Component

1. **a. Implement the state space for the chainbox.**

Solution:

Square Base (Position and Orientation):

The square base of the robot can move around in a 2D plane and rotate. To represent this, we use a space called **SE(2)**, which stands for "special Euclidean group in 2D." This space accounts for both the position (the x and y coordinates) and the rotation (the angle θ) of the base.

In OMPL, there's already a predefined space for this called `SE2StateSpace`, which combines the 2D position and the rotation.

Chain Links (4 Rotational Joints):

The 4-link chain attached to the base has 4 joints, each of which can rotate. Each joint has one degree of freedom, which means each joint can be represented by a circle of angles, or **S(1)**.

In OMPL, we represent each joint's angle using a space called `SO2StateSpace`. Since there are 4 joints, we need 4 `SO2StateSpace` objects, one for each joint.

Putting it All Together (Compound State Space):

To represent the entire robot (the base and the chain together), we need to combine these spaces. We do this using **CompoundStateSpace** in OMPL.

We combine the `SE2StateSpace` for the base and the 4 `SO2StateSpace` instances for the chain links into one big space that represents the full robot configuration.

This combined space looks like this:

$$SE(2) \times (SO(2))^4 = \mathbb{R}^2 \times (S^1)^5$$

Which means:

- The base moves in $\mathbb{R}^2$ (position) and rotates in S^1 (angle),
- The 4 chain joints rotate in S^1 (angles).

b. Implement the collision checking for the chainbox robot.

Solution:

Instead of using the provided `KinematicChain.h`, we created our own collision checking function to handle the chainbox robot's requirements. In our approach, the collision checker ensures that the robot maintains a valid state by checking collisions in the following ways:

(a) **Base vs. Obstacles**: We check each segment of the robot's square base to ensure it doesn't collide with any obstacles in the environment.

(b) **Chain vs. Obstacles**: Each segment of the 4-link chain is checked for collisions with obstacles, making sure it can move freely without intersecting with anything in the environment.

(c) **Chain vs. Box Base**: For all segments of the chain (except the first link, which is attached to the box), we check if they intersect with the square base. This ensures that the chain doesn't overlap with the base, except at the first joint.

(d) **Chain vs. Itself**: We also check if any of the chain's segments collide with each other, preventing self-intersections.

Collision checking function checks if two line segments intersect. It calculates vectors for both segments and then computes the determinant (`denom`) to check if they are parallel or collinear. If not, it uses numerators to determine if the segments overlap or intersect, returning `true` for intersection and `false` otherwise.

This way, our custom collision checking function ensures the robot stays within bounds and avoids collisions with obstacles, its own base, and itself.

2. a. Solve the narrow passage problem in Scenario 1.

Solution:

In the context of motion planning for narrow passage scenario, We opted for the **RRT-Connect** algorithm due to its inherent advantages over the other planners—RRT and PRM.

- The **RRT (Rapidly-exploring Random Tree) algorithm**, while effective for general exploration of high-dimensional spaces, often struggles with narrow passages. Its random sampling approach may lead to insufficient exploration in confined areas, making it less likely to find viable paths in such environments.
- On the other hand, the **Probabilistic Roadmap (PRM)** method is primarily beneficial for multi-query problems, where it constructs a roadmap of the environment. However, PRM also encounters challenges in narrow passages, as it relies on generating samples that may not adequately represent the constraints of these tight areas, necessitating a dense sampling strategy to increase success rates.
- In contrast, **RRT-Connect** employs a bidirectional search strategy, extending two trees—one from the start configuration and another from the goal configuration. This bidirectional approach significantly enhances its ability to efficiently navigate narrow passages. By growing trees towards each other, RRT-Connect effectively reduces the search space and increases the likelihood of finding a valid path through tight constraints. This characteristic is particularly advantageous in scenarios where the path needs to navigate around obstacles or through restricted areas, as it directs the search more purposefully.

Overall, RRT-Connect's strengths in efficiently handling narrow passages, combined with its ability to connect configurations from both the start and goal, make it the most suitable choice among the three planners for my specific motion planning problem.

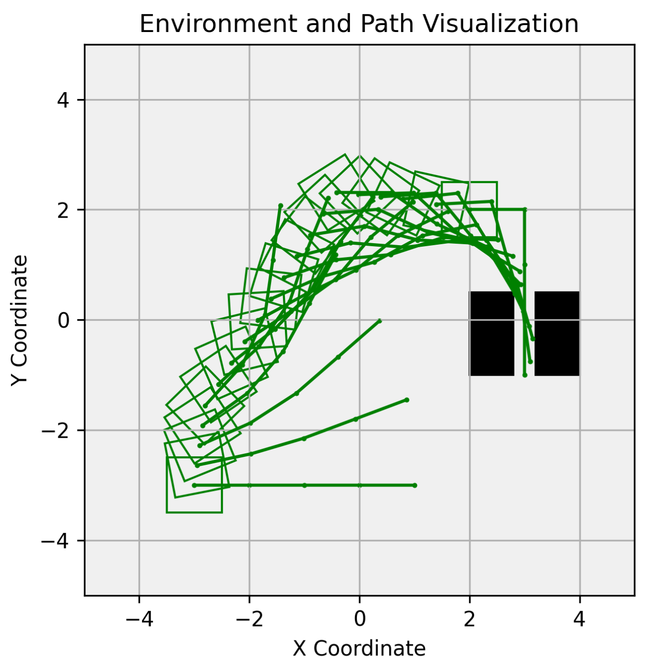


Figure 1: Chain box robot planning through narrow passage using RRT-Connect

b. Benchmark (Uniform, Gaussian, Bridge, and Obstacle)-based sampling for scenario1

Solution:

1. Time vs. Sampling Methods

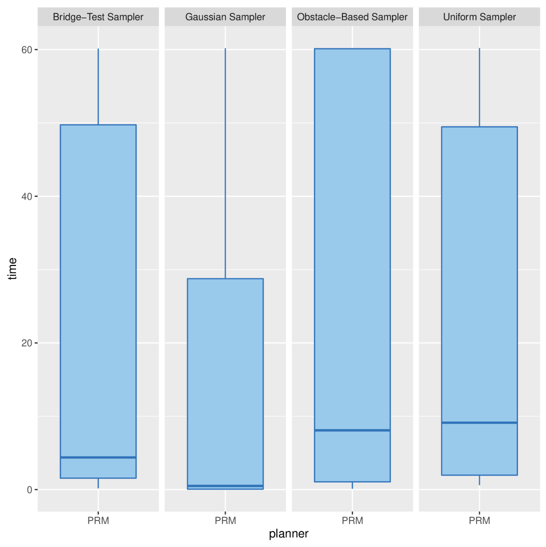


Figure 2: Time VS Different Sampling Methods using PRM

From the time comparison graph, we observe the following trends:

- **Bridge-Test Sampler**: This sampler took the moderate time to find a solution.
- **Gaussian Sampler**: This sampler completed in the shortest time, showing high efficiency.
- **Obstacle-Based Sampler** and **Uniform Sampler**: Both samplers have similar times, but are notably slower than the Gaussian Sampler and faster than the Bridge-Test Sampler.

2. Best Cost vs. Sampling Methods

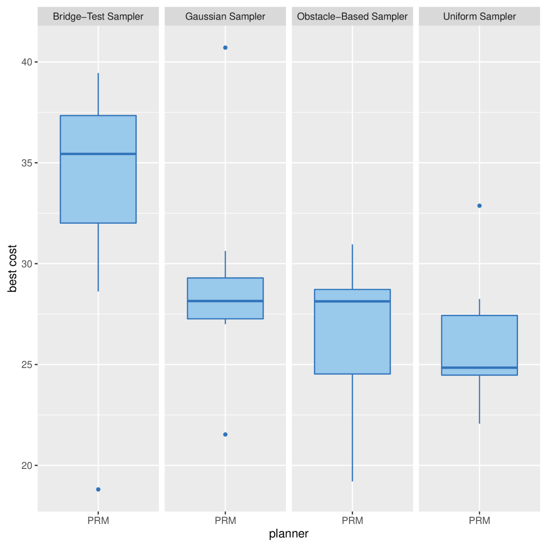


Figure 3: Best Cost VS Different Sampling Methods using PRM

From the best cost comparison, the performance is evaluated as follows:

- **Bridge-Test Sampler**: Has a relatively high best cost, indicating it finds less optimal solutions.
- **Gaussian Sampler**: Exhibits the lowest best cost, suggesting that it consistently finds more optimal paths.

- **Obstacle-Based Sampler**: Shows a wide variance in cost, but overall performs better than the Bridge-Test Sampler.
- **Uniform Sampler**: Similar to the Obstacle-Based Sampler with slightly better performance in cost but higher variance.

3. Graph States vs. Sampling Methods

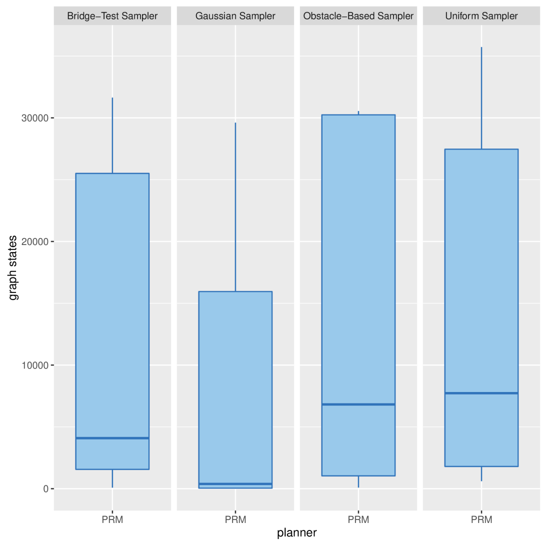


Figure 4: No. of States VS Different Sampling Methods using PRM

The number of graph states generated by each sampler is an indication of how efficiently it explores the state space:

- **Bridge-Test Sampler**: Generates a moderate number of graph states, indicating balanced exploration.
- **Gaussian Sampler**: Produces the fewest graph states, suggesting efficient space exploration and solution finding.
- **Obstacle-Based Sampler**: Produces the highest number of graph states, which may lead to increased computational load.
- **Uniform Sampler**: Also generates a high number of graph states, but with slightly lower variance compared to the Obstacle-Based Sampler.

4. Solution Length vs. Sampling Methods

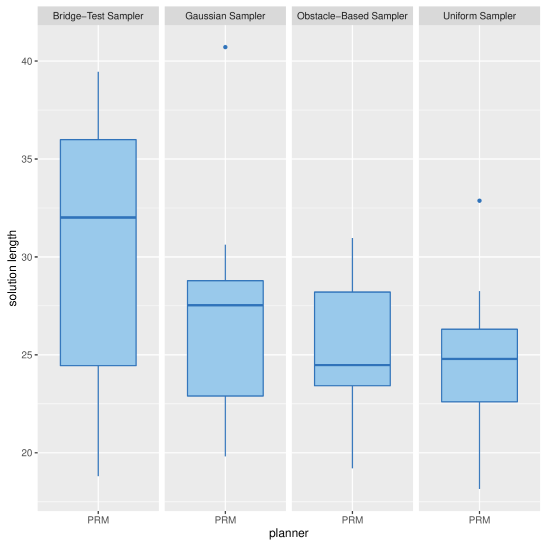


Figure 5: Solution Length VS Different Sampling Methods using PRM

The solution length generated by each sampler is an indication of how efficiently it finds a path to goal:

- **Bridge-Test Sampler**: Shows a wide covariance in solution length.
- **Gaussian Sampler**: Performs better than Bridge-Test Sampling, suggesting it mostly finds shorter paths than Bridge.
- **Obstacle-Based Sampler**: Produces the lowest solution length, making it the best among the other in terms of solution length.
- **Uniform Sampler**: Sometimes performs better than Obstacle-based sampling but the average is slightly higher than that of Obstacle-based sampling.

Conclusion
Based on the comparison between time, best cost, and graph states:

- The **Gaussian Sampler** performs the best overall, as it finds the most optimal solutions (lowest best cost) in the shortest time with the fewest graph states.
- The **Obstacle-Based Sampler** and **Uniform Sampler** also perform well but at the cost of generating more graph states and increased time.
- The **Bridge-Test Sampler** is the least efficient, taking the longest time and generating suboptimal solutions.

3. a. Solve Scenario2 by finding a path with maximum workspace clearance.

Solution:

In this scenario, we implemented a `ClearanceObjective` class in OMPL to calculate the cost of a state based on its clearance from obstacles. The clearance cost is defined as $\frac{1}{\text{min\_distance}}$, where min_distance is the minimum distance from the robot's center to the nearest obstacle corner. This class extends the `StateCostIntegralObjective`, allowing us to define a custom clearance metric using a combination of box and chain segments. We then designed the `planScenario2` function, which sets the optimization objective and employs the RRT* planner to find a path with maximum workspace clearance. RRT* is better suited for dynamically exploring complex spaces and optimizing paths based on continuous state costs, such as clearance from obstacles.

As we are not considering the chain links in our calculation of workspace clearance, it is important to note that even when the center of the box remains far from the obstacles and within the designated bounds, the behavior of the chain links differs significantly. In some cases, the chain links may extend outside the bounds and approach the obstacles, highlighting a discrepancy between the box's position and the chain's configuration. This observation underscores the need to account for the chain links when evaluating workspace clearance to ensure more accurate path planning.

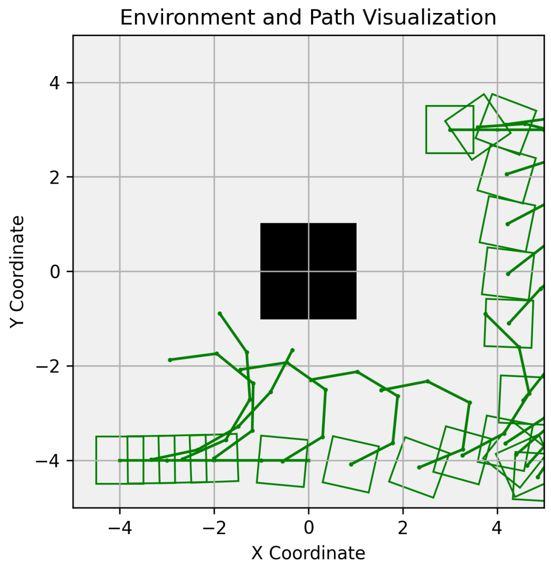


Figure 6: Box Chain robot fnding a path with maximum workspace clearance from center of box.

b. Benchmark rrt* and prm* using your the custom clearance objective.

Solution:

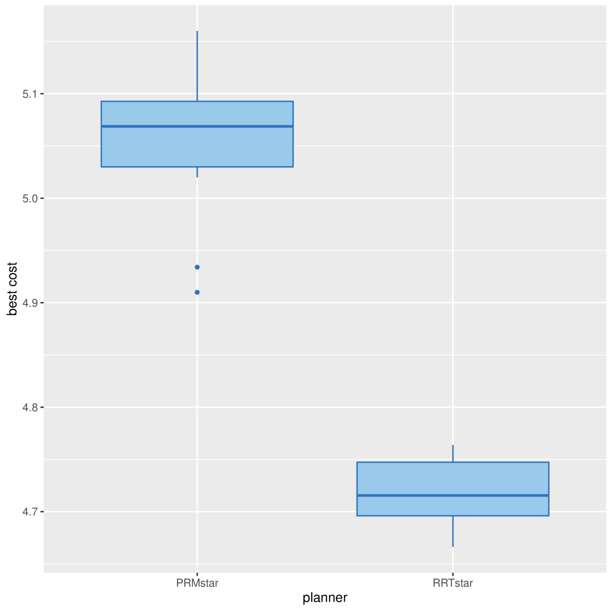


Figure 7: Best Cost vs RRT* and PRM*

Best Cost Comparison

(a) **Median best cost:** The median best cost for PRM$*$ is approximately 5.05, while RRT$*$ has a lower median cost of around 4.7, suggesting that RRT$*$ tends to find more cost-effective solutions.

(b) **Variability in cost:** PRM$*$ shows a larger interquartile range (IQR) than RRT$*$, indicating that the costs produced by PRM$*$ vary more widely compared to the more consistent costs found by RRT$*$.

(c) **Outliers:** PRM$*$ exhibits outliers at lower cost values, while no visible outliers are present for RRT$*$, reflecting more occasional extreme values in the PRM$*$ results.

(d) **Consistency of RRT$*$:** RRT$*$ generally achieves lower and more consistent costs compared to PRM$*$, making it a more reliable choice for optimizing cost.

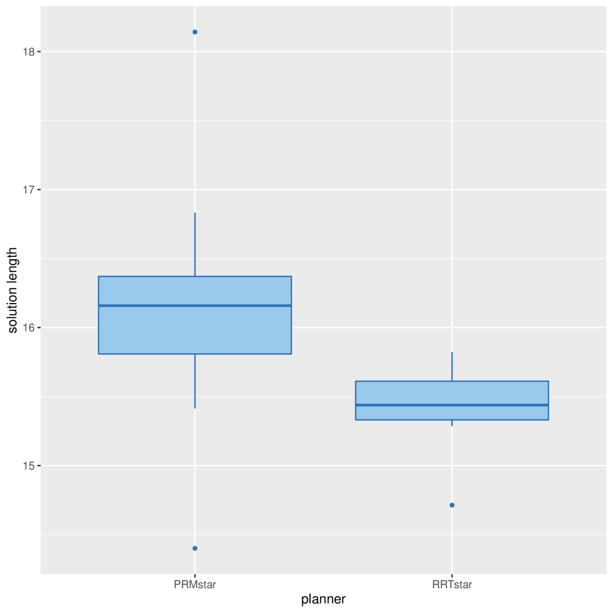


Figure 8: Solution Length vs RRT* and PRM*

Solution Length Comparison

(a) **Median solution length:** The median solution length for PRM∗ is slightly above 16.25, whereas RRT∗'s median solution length is around 15.5, indicating that RRT∗ typically finds shorter paths on average.

(b) **Variability in solution length:** PRM∗ exhibits a larger interquartile range (IQR), suggesting greater variability in the solution lengths it produces, compared to the more consistent lengths found by RRT∗.

(c) **Outliers:** PRM∗ has an outlier at approximately 18, while RRT∗ shows an outlier near 15, indicating occasional deviations from the typical solution length for both planners.

(d) **Consistency of RRT∗:** RRT∗ tends to find more consistent and generally shorter paths than PRM∗, which varies more in the length of the solutions it generates.

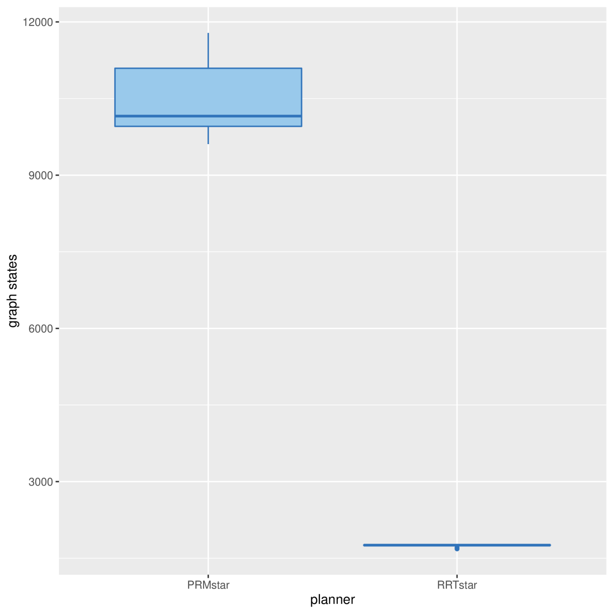


Figure 9: Graph States vs RRT* and PRM*

Graph States Comparison

(a) **PRM∗ consistently generates more graph states than RRT∗:** The box for PRM∗ lies significantly higher on the graph, with a range approximately between 9000 and 12000 graph states. This indicates that PRM∗ typically constructs a much larger graph, which could be a result of its sampling strategy that emphasizes global exploration of the space.

(b) **RRT∗ generates fewer graph states with minimal variance:** RRT∗ shows a much smaller range, with graph states clustering near 3000. This suggests that RRT∗ tends to use fewer nodes in its planning process. The narrow range indicates lower variability in its performance for this metric.

(c) **Variance in PRM∗:** PRM∗ exhibits more variability in the number of graph states, as indicated by the wider box, which spans from around 9000 to 12000. This suggests that the performance of PRM∗ in terms of graph size may depend on the specific environment or the random samples drawn.

(d) **Implication of graph size:** A larger number of graph states in PRM∗ could indicate better exploration but at the cost of higher computational overhead. In contrast, RRT∗'s smaller graph size suggests that it may offer faster computation, but it might not explore the space as thoroughly as PRM∗.

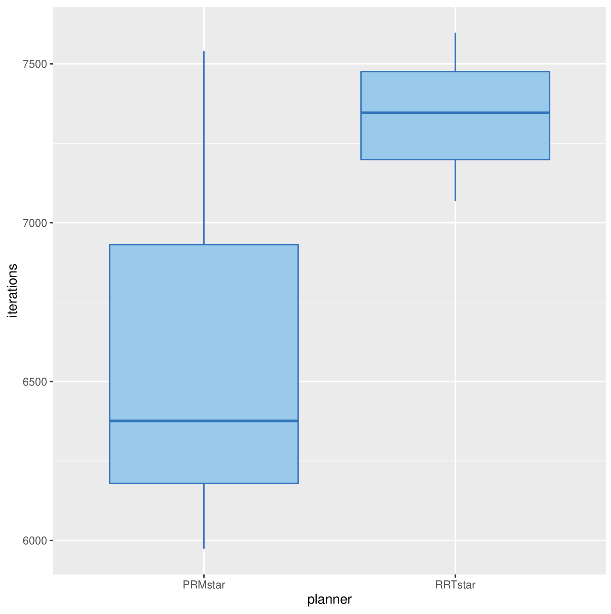


Figure 10: Iterations vs RRT* and PRM*

Iterations Comparison

(a) **PRM∗ shows greater variability in the number of iterations:** The boxplot for PRM∗ spans a wide range from approximately 6000 to 7500 iterations, with a median around 6500. This high variance suggests that PRM∗ performs inconsistently in terms of iterations, depending on the specific circumstances or problem setup.

(b) **RRT∗ is more consistent:** RRT∗ has a much narrower boxplot, with iterations ranging from around 7000 to 7500. This indicates more consistent performance, with less fluctuation across different runs of the algorithm.

(c) **Comparison of median iterations:** The median value for PRM∗ (approximately 6500 iterations) is lower than that of RRT∗, which has a median around 7200. This implies that PRM∗ may complete certain tasks with fewer iterations on average compared to RRT∗, although its performance varies more significantly.

(d) **Outliers in PRM∗:** The PRM∗ plot includes outliers that extend beyond 7500 iterations, suggesting that in certain scenarios, PRM∗ may require a significantly higher number of iterations, likely due to difficulties in finding a solution.

Conclusion
RRT* has better consistency and cost, lower solution length and graph states, and also does more iterations in same time than PRM*.

4. **Bonus: Implement the true workspace clearance function, which calculates the minimum distance between the entire chainbox robot and the obstacles. Explain how you implemented this in your report. Submit the solution as optimal_clear.gif and optimal_clear_path.txt**

Solution:

For this part, we have modified `plan.cpp` such that it asks for 3 inputs where input of `(1) Robot Reaching Task`, `(2) Robot Avoiding Task` and `(3) (BONUS) Robot Avoiding Task`.

Similar to Question 3, we implemented a `ClearanceObjective` class in OMPL to compute the cost of a state based on its clearance from obstacles. In this case, we calculate the distance between each vertex of the box and each vertex of the obstacles. Additionally, we assess the distance between each vertex of the chain links and each vertex of the obstacles. By determining the minimum distance (clearance) between all the vertices of the robot and the obstacles, we derive the overall path cost.

$$PathCost \propto \frac{1}{clearance}$$

We then modified the `planScenario2` function, which sets the optimization objective and utilizes the RRT* planner to identify a path that maximizes workspace clearance. After generating the solution, we simplified and validated the path, saving it to a file to improve its efficiency. To ensure the planner produces a desirable and optimal solution, we also adjusted the `setRange` and `GoalBias` parameters for RRT*. Additionally, we incorporated a heuristic to control the rotation of the box-chain robot, prioritizing rotation towards the goal position before advancing in the x and y coordinates.

As we can see in the image below, not only box base but also the chain links plan from start to goal position with maximum workspace clearance possible. For this plan we were able to achieve the **path cost** of **16.124**.

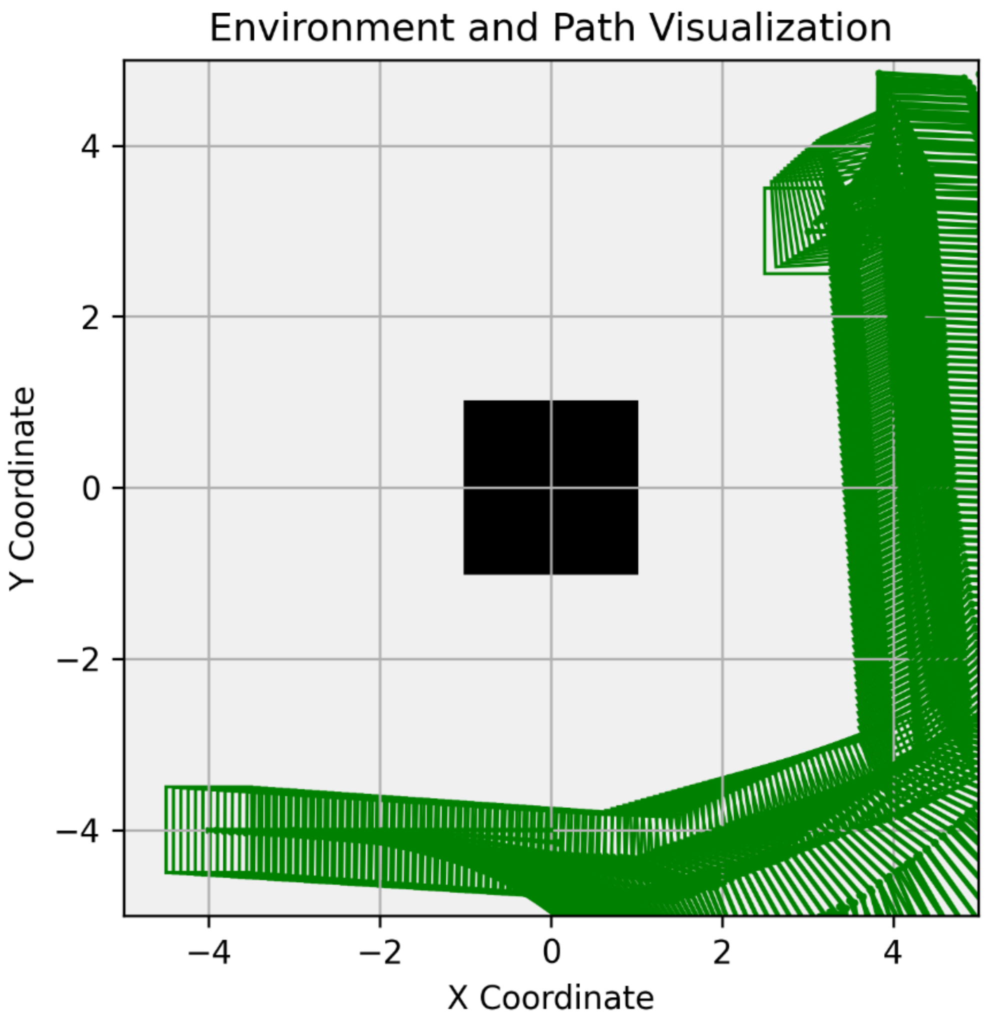


Figure 11: Box Chain robot finding a path with maximum workspace clearance from entire chain box robot .