RBE550: Motion Planning

Motion Planning HW2

Student 1: Harsh Chhajed
Student 2: Pranay Katyal

Q1 : Comparing the Bug 1 and Bug 2 Algorithms.

Bug 1 Algorithm

Assumptions of Bug 1

- We know the direction towards the goal.
- Awareness about surrounding is there, like walls and obstacles.
- Must have memory so it doesn't get stuck in loops like Bug 0.
- The world should be reasonable, that is workspace has limits, and we have finite number of obstacles.

Bug 1 Strategy

- Head Towards the goal.
- If an obstacle is encountered, circumnavigate it and remember how close you get to the goal.
- Then return to the closest point, by wall following and continue.

Bug 1 Algorithm

Algorithm 1 Bug 1

Require: A point robot with a tactile sensor
Ensure: A path to q_{goal} or a conclusion that no such path exists

$q^{L_0} \leftarrow q_{start}$
$i \leftarrow 1$
repeat
  repeat
    Move from $q^{L_{i-1}}$ towards q_{goal}
  until q_{goal} is reached or obstacle encountered at q^{H_i}
  if q_{goal} is reached **then**
    Exit
  repeat
    Follow the obstacle boundary while recording q^{L_i} with the shortest distance to q_{goal}
  until q_{goal} is reached or q^{H_i} is re-encountered
  if q_{goal} is reached **then**
    Exit
  Move to q^{L_i}
  if moving toward q_{goal} results in an obstacle **then**
    Exit with Failure
  else
    $i \leftarrow i + 1$
    Continue
until termination condition is met (q_{goal} is found)

- q^{H_i} is the Hit Point.
- q^{L_i} is the Leave Point.

Bug 1 Bounds

- The Lower bound for Bug 1 is D.
- The Upper bound for Bug 2 is $D + (1.5) * \sum P_i$.

Bug 2 Algorithm

Assumptions of Bug 2

- We know the direction towards the goal.
- Awareness about surrounding is there, like walls and obstacles.
- Must have memory so it doesn't get stuck in loops like Bug 0.
- The world should be reasonable, that is workspace has limits, and we have finite number of obstacles.

Bug 2 Strategy

- Head Towards the goal.
- If an obstacle is encountered, circumnavigate around it, until it encounters the m-line again.
- Leave the obstacle and continue.

Bug 2

Algorithm 2 Bug 2 Algorithm

Require: A point robot with a tactile sensor
Ensure: A path to q_{goal} or a conclusion that no such path exists

Let $q^{L_0} = q_{start}$, $i = 1$
repeat
  repeat
    Move from $q^{L_{i-1}}$ toward q_{goal} along the m-line
  until q_{goal} is reached or an obstacle is encountered at q^{H_i}
  if q_{goal} is reached **then**
    return "Path found"
  repeat
    Follow the obstacle boundary
  until q_{goal} is reached or q^{H_i} is re-encountered or m-line is re-encountered,
    $x \neq q^{H_i}$, $d(x, q_{goal}) < d(q^{H_i}, q_{goal})$ and the way to goal is unimpeded
  if q_{goal} is reached **then**
    return "Path found"
  if q^{H_i} is reached **then**
    return "Path not found"
  else
    $i \leftarrow i + 1$
    continue
until termination condition is met

Bug 2 Bounds

- The Lower bound for Bug 2 is D.
- The Upper bound for Bug 2 is $D + (0.5) * \sum n_i P_i$.

Comparison of Bug 1 and Bug 2

Bug 1 Features

- Bug 1 is a Complete Algorithm, that is, if Solution exists, Bug 1 while return it, if it doesn't, Bug 1 Exits with Failure. It doesn't get stuck in an infinite search loop.
- Bug 1 is an Exhaustive search Algorithm, It looks at all the choices before committing.

- Bug 1 has a more predictable overall performance.
- Bug 1 is Safe and Reliable.

Bug 2 Features

- Bug 2 is a Greedy Algorithm, It takes the first path that looks better.
- Bug 2 is Overall better in some cases, but worse in others cases than Bug 1.

Q2

(a) Admissible and Non-Admissible Heuristics in Grid World

Admissible Heuristic h_a

In a grid world where an agent can move up, down, left, right, and diagonally (with all moves costing 1), an admissible heuristic is the Chebyshev distance:

$$h_a(x, y) = \max(|x - g_x|, |y - g_y|)$$

Explanation

Since diagonal moves are allowed and cost the same as horizontal/vertical moves, the shortest path behaves like a king's movement (or Queen too) in chess. The Chebyshev distance gives the minimum number of moves required to reach the goal. It never overestimates the true cost $T(n)$, ensuring admissibility.

Example

Consider an agent at position $(2, 3)$ and a goal at $(5, 7)$:

$$h_a(2, 3) = \max(|2 - 5|, |3 - 7|) = \max(3, 4) = 4$$

If the true shortest path consists of 3 diagonal moves and 1 vertical move, the actual cost is also 4, making h_a exact.

Case Where h_a Underestimates

Consider an agent at $(2, 3)$ and a goal at $(6, 7)$:

$$h_a(2, 3) = \max(|2 - 6|, |3 - 7|) = \max(4, 4) = 4$$

If the best path involves an additional step in a horizontal or vertical direction before making diagonal moves (maybe due to some obstacle), then the true cost could be 5, meaning h_a is admissible and is slightly underestimating the True Cost.

Non-Admissible Heuristic h_b

A non-admissible heuristic would be the Euclidean distance:

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

Explanation

This heuristic represents the straight-line distance between the agent and the goal. However, since the agent moves in din steps of weight 1 whether moving horizontally, vertically, or Diagonally, the Euclidean distance can sometimes overestimate the true cost, violating admissibility.

Example of Overestimation

Using the same positions $(2, 3)$ and $(5, 7)$:

$$h_b(2,3) = \sqrt{(2-5)^2 + (3-7)^2} = \sqrt{9+16} = 5$$

If the optimal path allows diagonal moves and has a true cost of 4, then $h_b = 5$ overestimates, making it non-admissible.

Modification to Make Euclidean Admissible

To make the Euclidean heuristic admissible, we can scale it down (here scaling down by 10):

$$h'_E(x,y) = \frac{\sqrt{(x-g_x)^2 + (y-g_y)^2}}{10}$$

This may ensure it will never overestimate the true path cost,thereby making it admissible.

(b) Proving Consistency of $h(n) = \max(h_1(n), h_2(n))$

Given: h_1 and h_2 are consistent heuristics.

Consistency Definition: A heuristic h is consistent if for every node n and its neighbor n':

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

where $T(n, n')$ is the true cost from n to n'.

Proof:

1. **Let us start with the definition of $h(n)$:**

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

2. **Applying consistency for h_1 and h_2:**
Since h_1 and h_2 are consistent:

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

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

3. **Taking the maximum of both inequalities:**

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

4. **Factoring out the common term $T(n, n')$:**
Since $T(n, n')$ is common in both terms:

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

5. **Substitute back the definition of $h(n)$:**

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

Conclusion: Since $h(n) = \max(h_1(n), h_2(n))$ satisfies the consistency condition, $h(n)$ is consistent.

Defining the Heuristics

We define two heuristic functions that align with the given movement costs:

- **Heuristic $h_1(n)$:** The *Chebyshev distance*, which is the maximum coordinate difference:

$$h_1(x, y) = \max(|x - g_x|, |y - g_y|)$$

  This heuristic is appropriate when diagonal moves have a uniform cost of .

- **Heuristic $h_2(n)$:** A *weighted Chebyshev distance* to provide another admissible and consistent function:

$$h_2(x, y) = \frac{|x - g_x| + |y - g_y|}{2}$$

  This heuristic is also admissible, as it never overestimates the true cost.

Example Calculation

We consider a scenario with:

- Start node: $n = (2, 3)$
- Neighbor node: $n' = (3, 3)$
- Goal node: $g = (5, 7)$
- Movement cost: $T(n, n') = 1$ (horizontal, vertical, or diagonal move)

Computing Heuristic Values

For the start node $n = (2, 3)$:

$$h_1(2, 3) = \max(|2 - 5|, |3 - 7|) = \max(3, 4) = 4$$

$$h_2(2, 3) = \frac{|2 - 5| + |3 - 7|}{2} = \frac{3 + 4}{2} = 3.5$$

$$h(2, 3) = \max(h_1(2, 3), h_2(2, 3)) = \max(4, 3.5) = 4$$

For the neighbor node $n' = (3, 3)$:

$$h_1(3, 3) = \max(|3 - 5|, |3 - 7|) = \max(2, 4) = 4$$

$$h_2(3, 3) = \frac{|3 - 5| + |3 - 7|}{2} = \frac{2 + 4}{2} = 3$$

$$h(3, 3) = \max(h_1(3, 3), h_2(3, 3)) = \max(4, 3) = 4$$

Checking Consistency Condition

A heuristic $h(n)$ is consistent if it satisfies the condition:

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

Substituting the computed values:

$$4 \leq 1 + 4 = 5$$

Since this inequality holds, the heuristic function $h(n)$ is consistent.

Conclusion

We have demonstrated that the heuristic function:

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

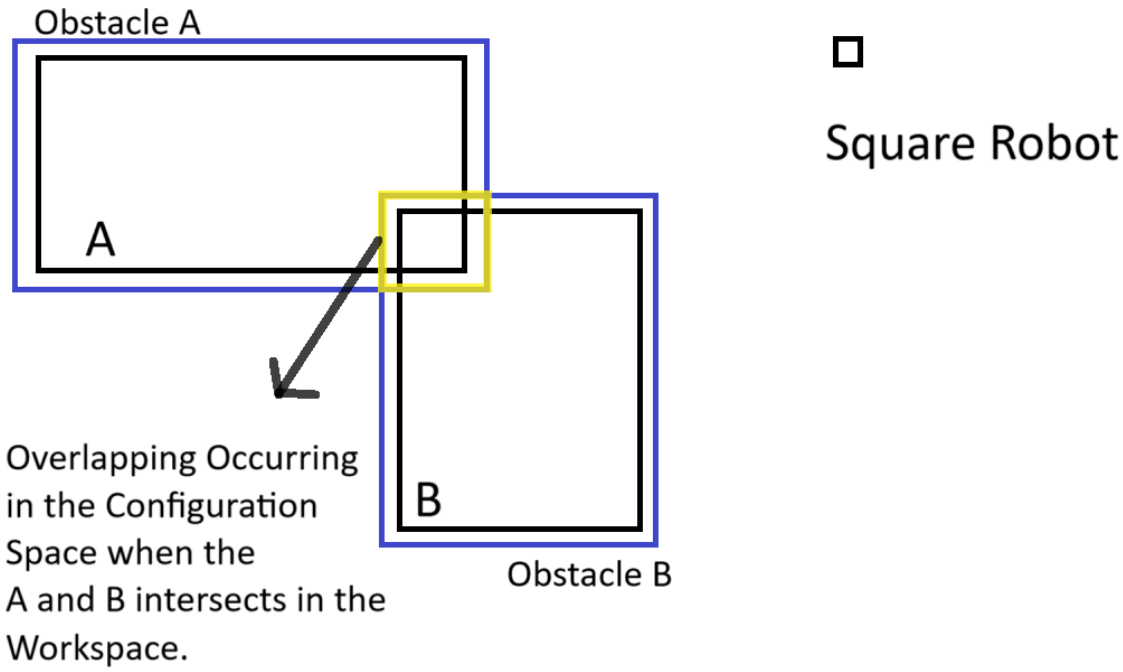
is both admissible and consistent for path planning problems.

Q3

Configuration Space Obstacle Overlaps

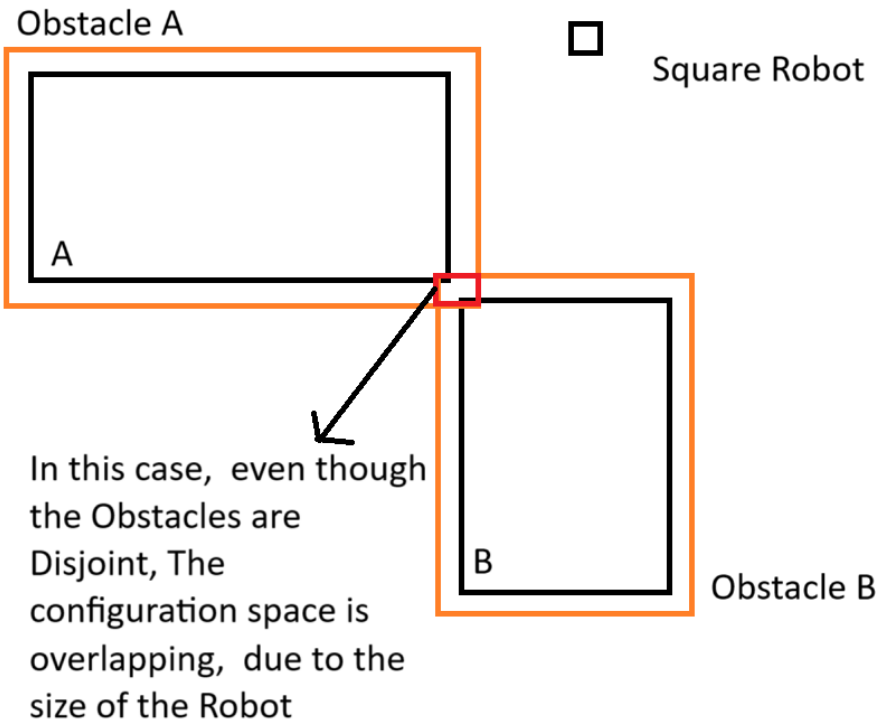
Question 1: If the workspace obstacles intersect ($A \cap B \neq \emptyset$), do the configuration space obstacles Q_A and Q_B always overlap?

Explanation: Yes. When workspace obstacles A and B intersect, their configuration space representations Q_A and Q_B must also overlap. This is because the configuration space is constructed by inflating each workspace obstacle by the shape of the robot, treated as a point. If any point x exists in both A and B, it implies that the robot configurations leading to a collision at x will intersect in both Q_A and Q_B. Therefore, the intersection of A and B in the workspace leads to an overlap in their configuration space representations.



Question 2: If the workspace obstacles are disjoint ($A \cap B = \emptyset$), is it possible for the configuration space obstacles Q_A and Q_B to overlap?)

Explanation: Yes, it is possible for Q_A and Q_B to overlap despite A and B being disjoint in the workspace. This potential overlap arises from the method by which the configuration space is defined, particularly through the Minkowski difference which inflates each obstacle by the shape of the robot. If the gap between A and B is smaller than the robot's size, the configuration space representations Q_A and Q_B can overlap. This scenario shows how even disjoint obstacles in the workspace can affect the robot's ability to navigate between them due to the increased area they cover in the configuration space.



Q4 : Visibility Graph Construction and Path Planning

Visibility Graph Complexity

Algorithm for Constructing the Visibility Graph

- The visibility graph algorithm involves examining every pair of vertices and checking if a direct line between them intersects any edge of obstacle.
- It is given that the intersection check of two line segments can be done in constant time.
- The total number of vertex pairs is combinatorial, leading to a complexity of $O(n^3)$.
- The below algorithm loops over each pair of vertices and checks if the line segment between them intersects with any obstacle edge. Given there are n vertices, checking every pair implies $n * (n-1)/2$ checks. For each pair, checking against all obstacle edges, which could in the worst case be O(n), leads to an overall complexity of $O(n^3)$.

Pseudocode for Visibility Graph Construction

Algorithm 3 Construct Visibility Graph

Require: Set of vertices V including obstacle vertices (O) and start (s) and goal (g) points
Ensure: A visibility graph G where each edge (v, w) represents a direct line of sight

```
Initialize an empty graph G
V ← V ∪ {s, g}
for each vertex v ∈ V do
    for each vertex w ∈ V where w ≠ v do
        if NoObstacleBetween(v, w , O) then
            Add edge (v, w) to graph G
return G
```

Function to Check Obstacles Between Two Points

Algorithm 4 NoObstacleBetween

Require: Two vertices v and w, Set of obstacles O
Ensure: Boolean indicating if the line segment between v and w is free of obstacles

```
for each edge e in O do
    if Intersects(LineSegment(v, w), e) then
        return false
return true
```

Path Planning Using the Visibility Graph

Using the Visibility Graph for Path Planning

- The visibility graph can be used to find the shortest path from the start to the goal point.
- Algorithms like Dijkstra's or A* can be used to compute the shortest path.
- The complexity of Dijkstra's algorithm in terms of vertices (n) and edges (m) in the graph is O(m + n log n).

Pseudocode for Dijkstra's Algorithm

Algorithm 5 Dijkstra's Algorithm for Shortest Path

Require: A visibility graph, start vertex, and goal vertex
Ensure: The shortest path from start to goal if exists

```
Initialize distance of all vertices as infinity except start vertex as 0
Initialize a priority queue with (0, start vertex)
while priority queue is not empty do
    current_distance, current_vertex = extract-min from priority queue
    if current_vertex is goal then
        return current_distance
    for each neighbor of current_vertex in graph do
        if current_distance + edge_weight ¡ distance[neighbor] then
            distance[neighbor] = current_distance + edge_weight
            Add (distance[neighbor], neighbor) to priority queue
return infinity // goal is not reachable
```

Pseudocode for A* Algorithm

Algorithm 6 A* Algorithm for Shortest Path

Require: A visibility graph G, start vertex s, goal vertex g, heuristic function h
Ensure: The shortest path from s to g if exists

Initialize *openSet* with start vertex s
cameFrom is an empty map to store the path
$gScore[v]$ is a map with default value of infinity for all v except $gScore[s] = 0$
$fScore[v]$ is a map with default value of infinity for all v except $fScore[s] = h(s, g)$
while not *openSet* is empty **do**
    $current$ = vertex in *openSet* with the lowest $fScore[current]$
    if $current = g$ **then**
        return ReconstructPath($cameFrom, current$)
    Remove *current* from *openSet*
    for each neighbor of *current* in G **do**
        $tentative\_gScore = gScore[current] + \text{dist}(current, neighbor)$
        if $tentative\_gScore < gScore[neighbor]$ **then**
            $cameFrom[neighbor] = current$
            $gScore[neighbor] = tentative\_gScore$
            $fScore[neighbor] = gScore[neighbor] + h(neighbor, g)$
            if neighbor is not in *openSet* **then**
                Add neighbor to *openSet*
return "Failure, path not found"

Function to Reconstruct Path

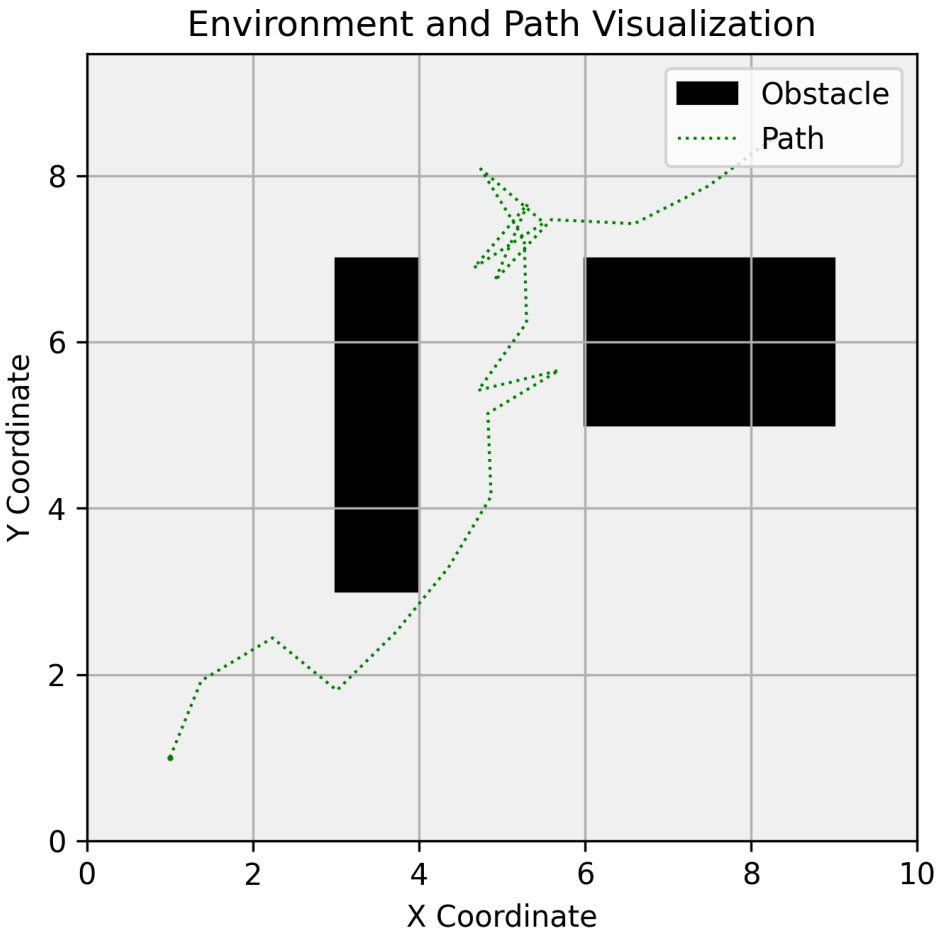
Algorithm 7 ReconstructPath

Require: *cameFrom*, current node *current*
Ensure: Path from start to current node

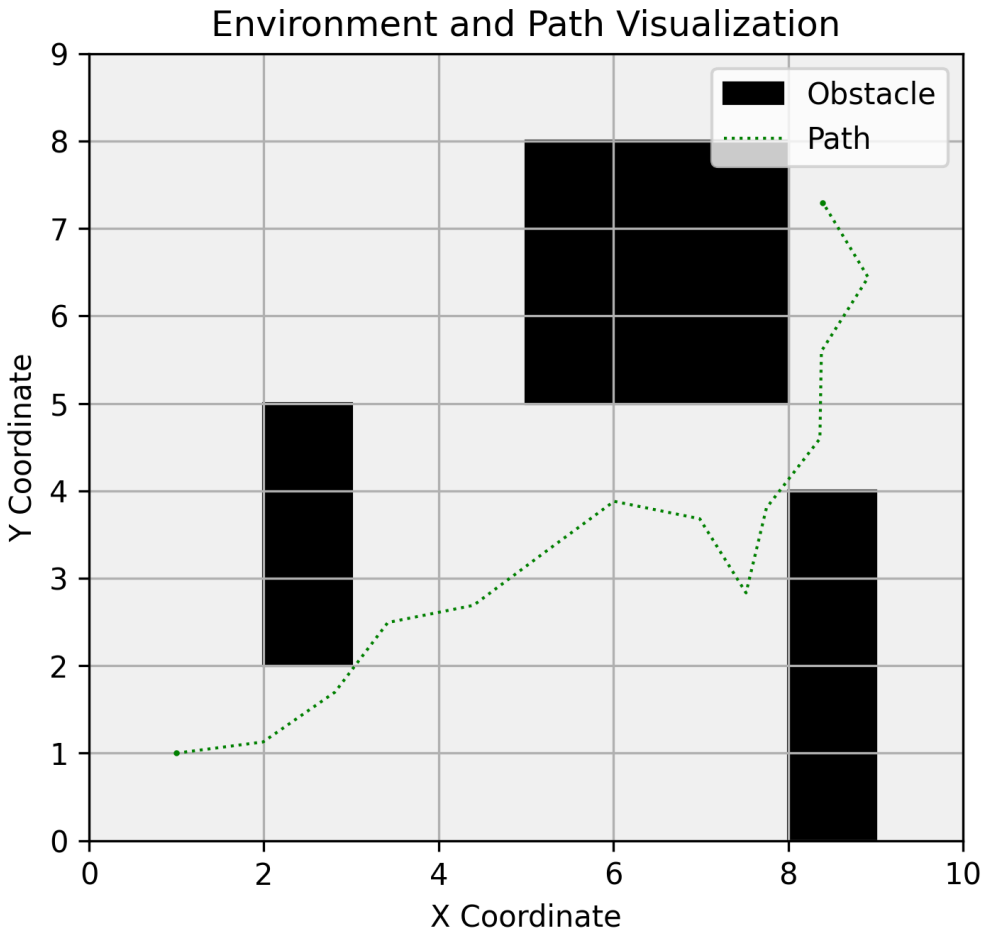
$total\_path$ = [current]
while *current* is in $cameFrom.keys()$ **do**
    $current = cameFrom[current]$
    Prepend *current* to $total\_path$
return $total\_path$

Coding Part Images

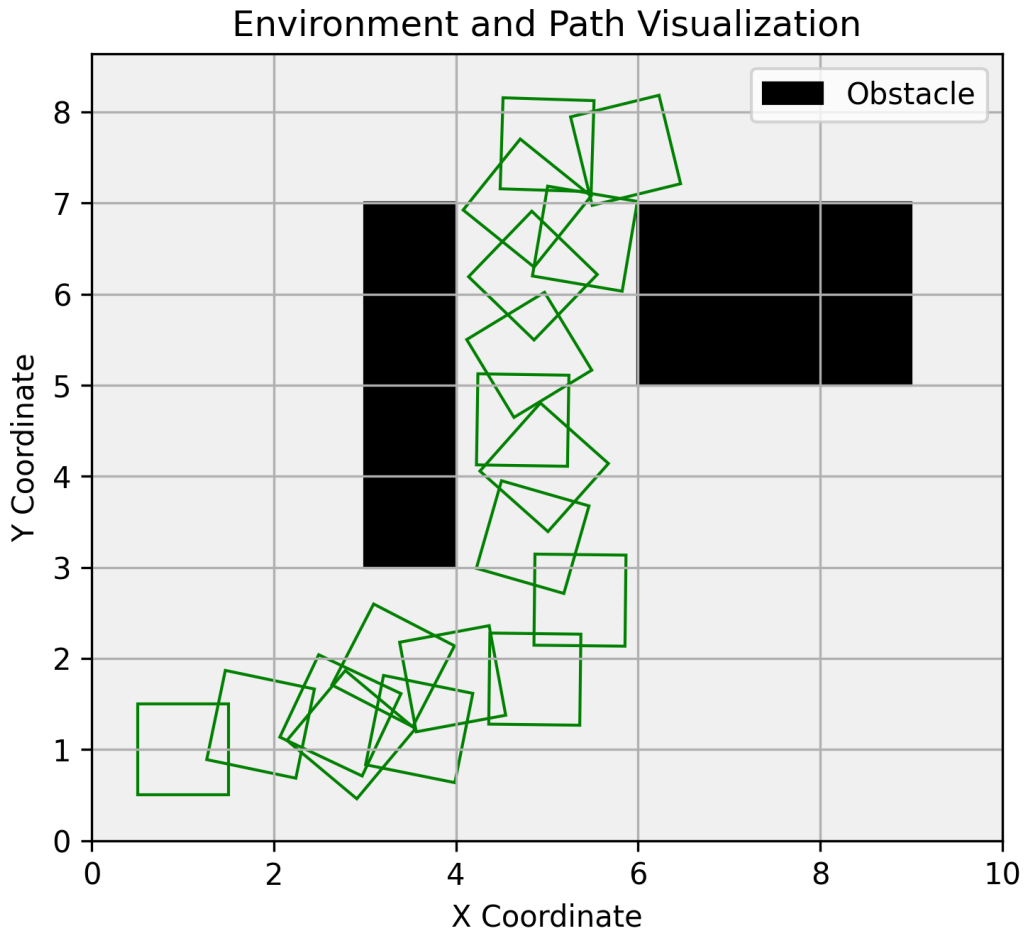
Point Robot in Environment 1



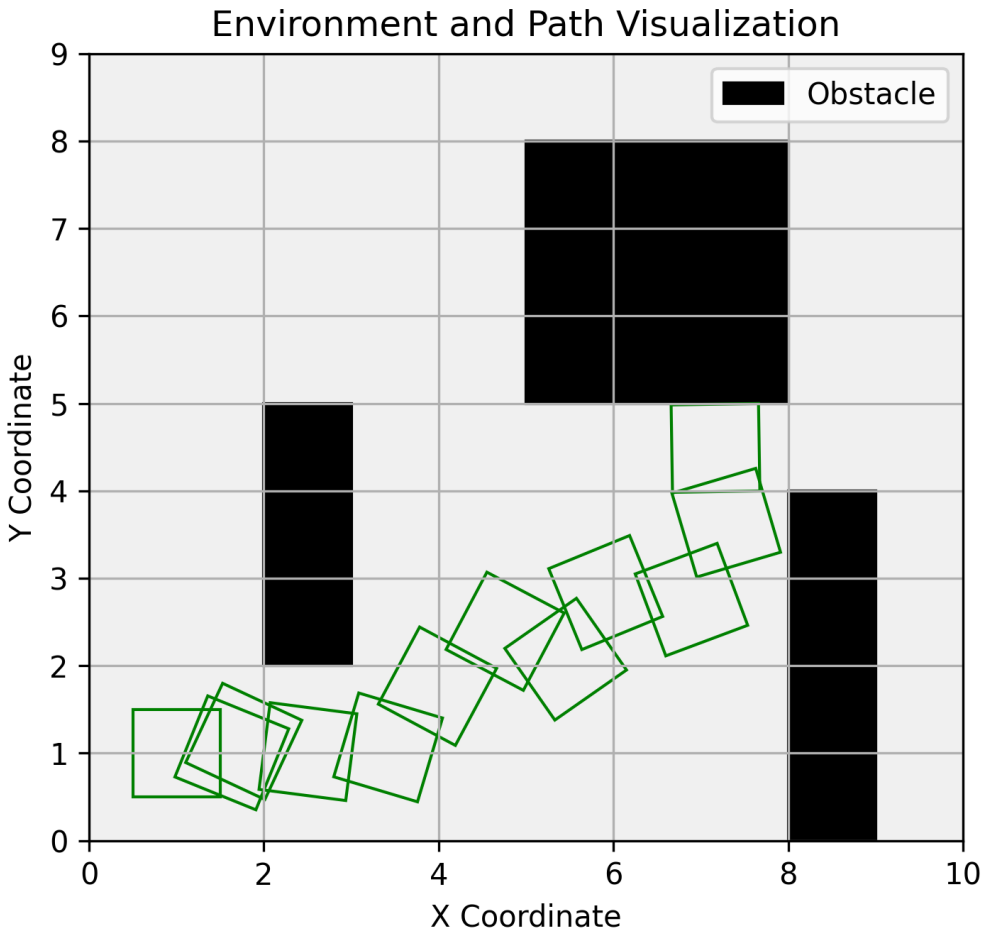
Point Robot in Environment 2



Box Robot in Environment 1



Box Robot in Environment 2



Obstacle Dimensions for both Environments

```
void makeEnvironment1(std::vector<Rectangle> &obstacles)
{
    // Example: Two obstacles.
    obstacles.push_back({3.0, 3.0, 1.0, 4.0});
    obstacles.push_back({6.0, 5.0, 3.0, 2.0});
}

void makeEnvironment2(std::vector<Rectangle> &obstacles)
{
    // Example: Three obstacles.
    obstacles.push_back({2.0, 2.0, 1.0, 3.0});
    obstacles.push_back({5.0, 5.0, 3.0, 3.0});
    obstacles.push_back({8.0, 0.0, 1.0, 4.0});
}
```

Point Robot, Start and Goal Queries

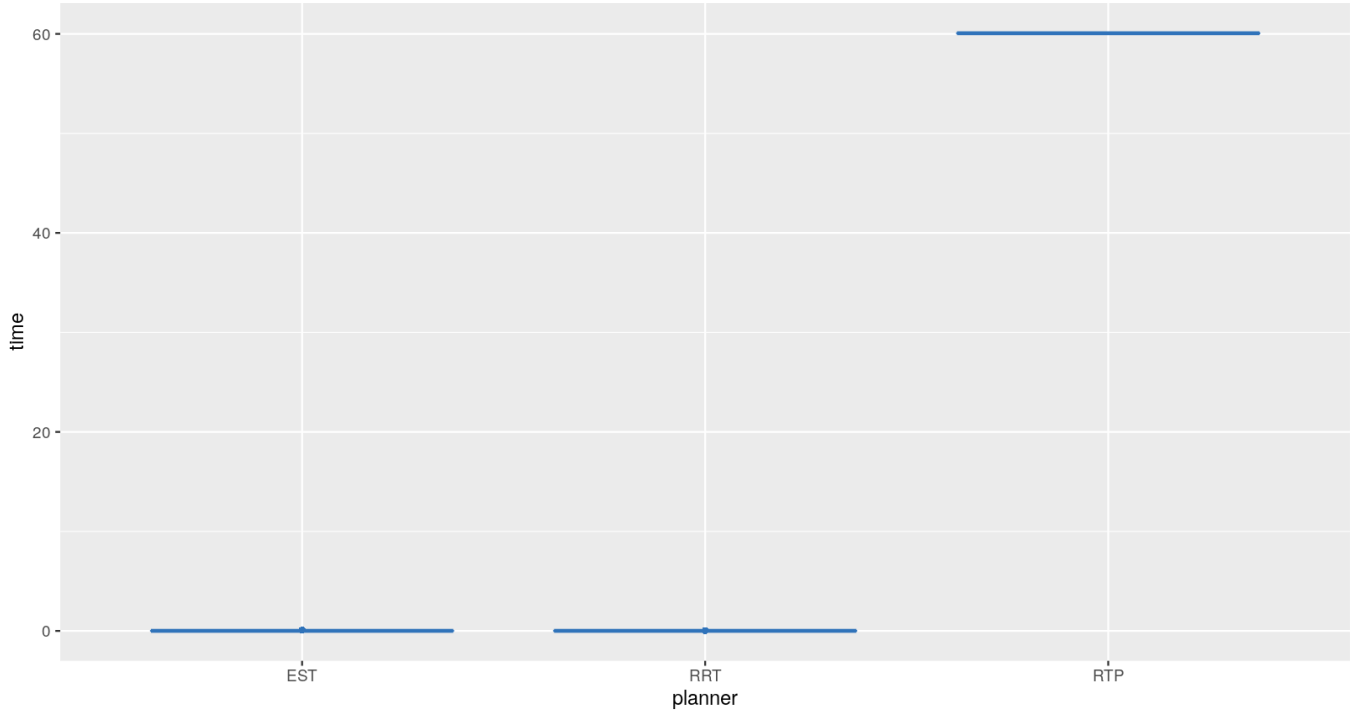
```
// Define start and goal states.
ob::ScopedState<> start(space);
start[0] = 1.0;
start[1] = 1.0;
ob::ScopedState<> goal(space);
goal[0] = 9.0;
goal[1] = 9.0;
```

Box Robot, Start and Goal Queries

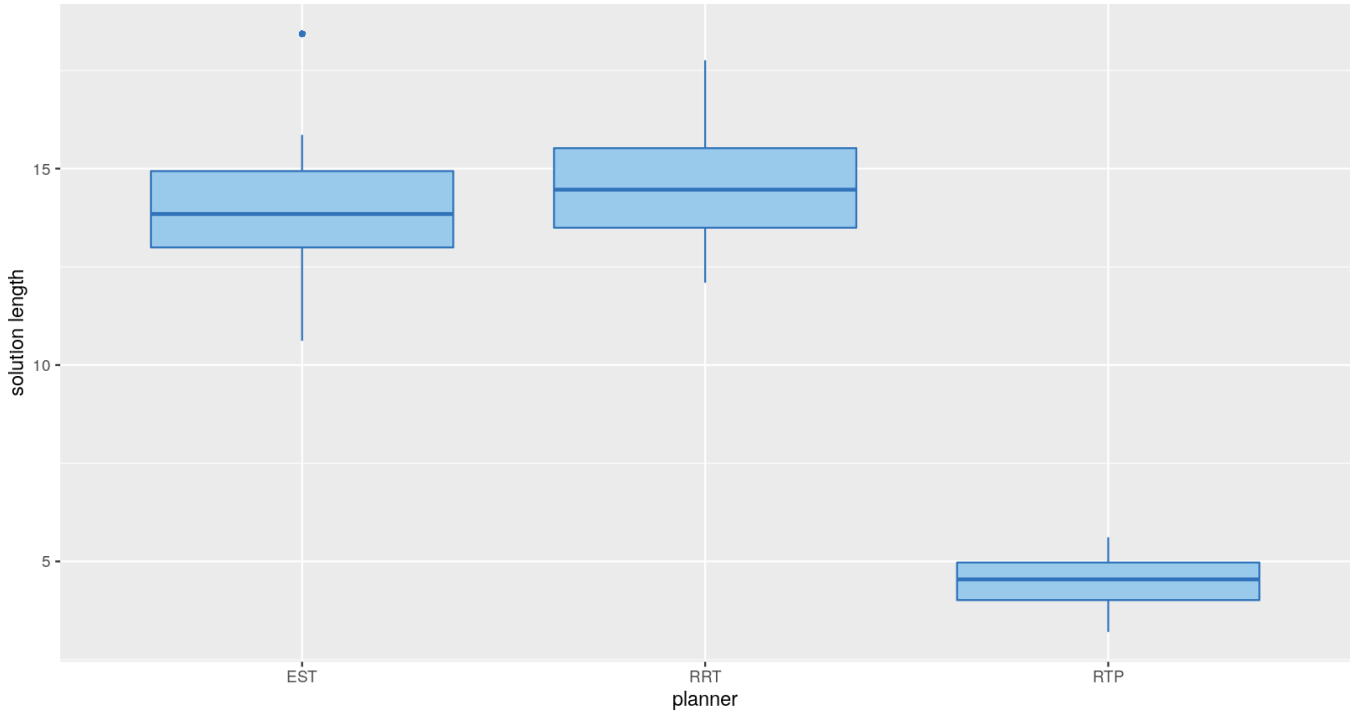
```
// Define start and goal states.
ob::ScopedState<ob::SE2StateSpace> start(space);
start->setX(1.0);
start->setY(1.0);
start->setYaw(0.0);
ob::ScopedState<ob::SE2StateSpace> goal(space);
goal->setX(9.0);
goal->setY(9.0);
goal->setYaw(0.0);
```

Benchmarking Plots for 10 Linked Chain

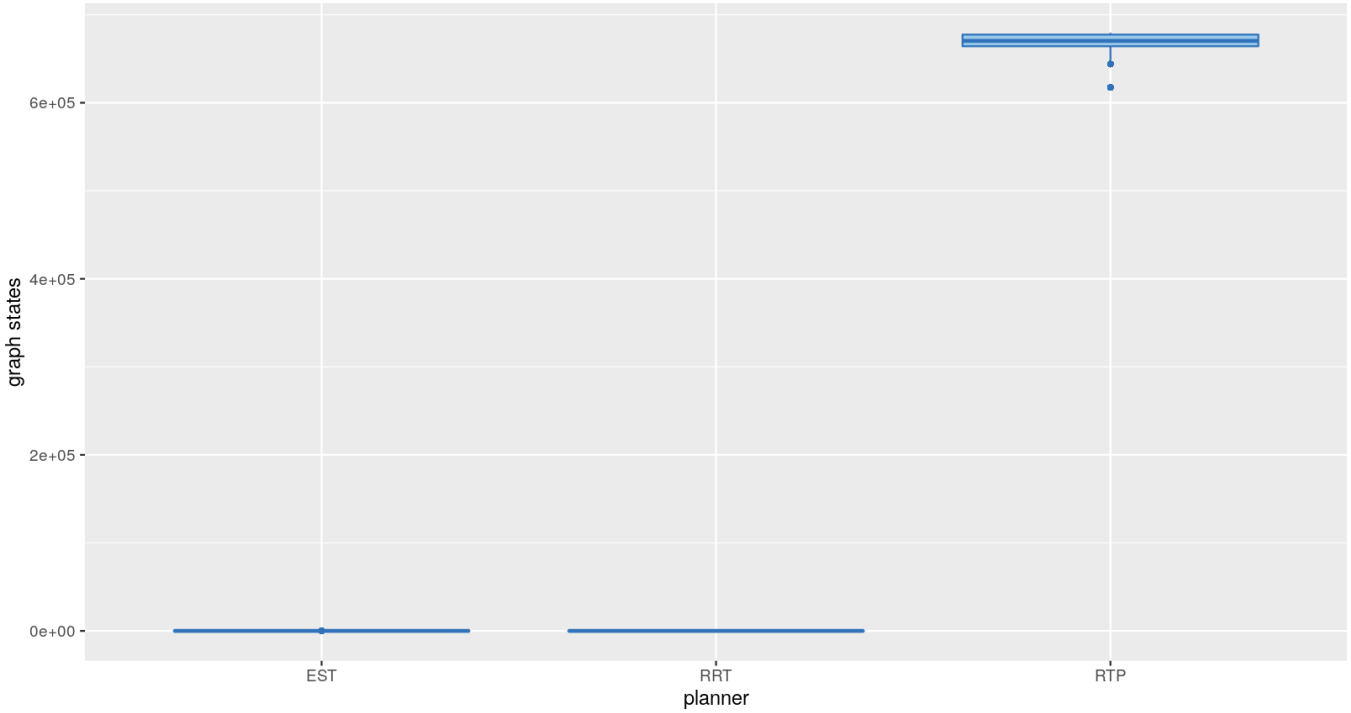
Time Benchmarking



Solution Length Benchmarking

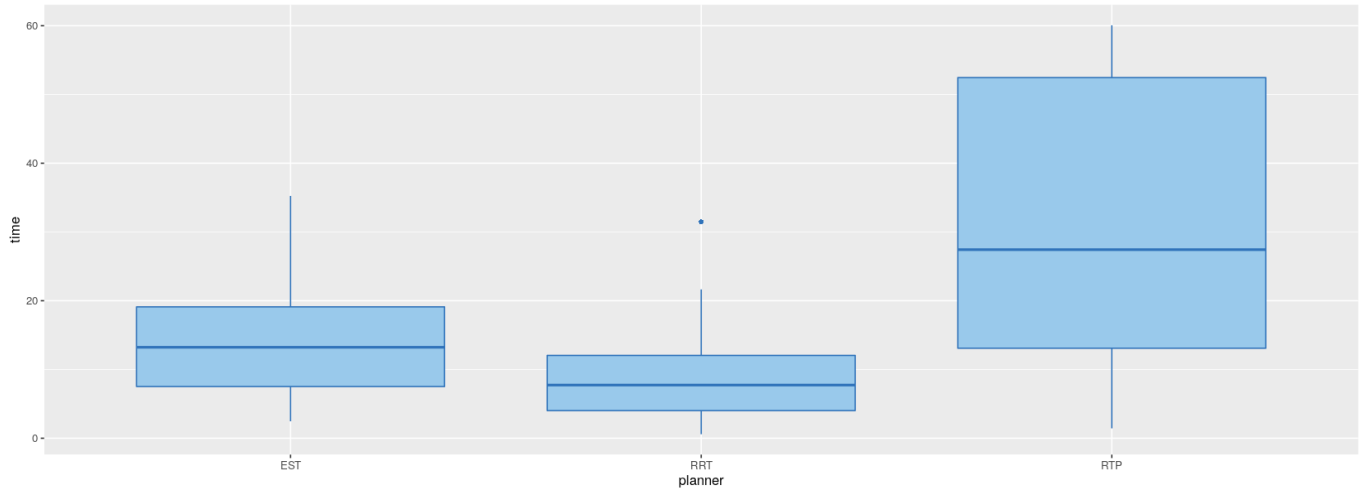


Graph state Count Benchmarking

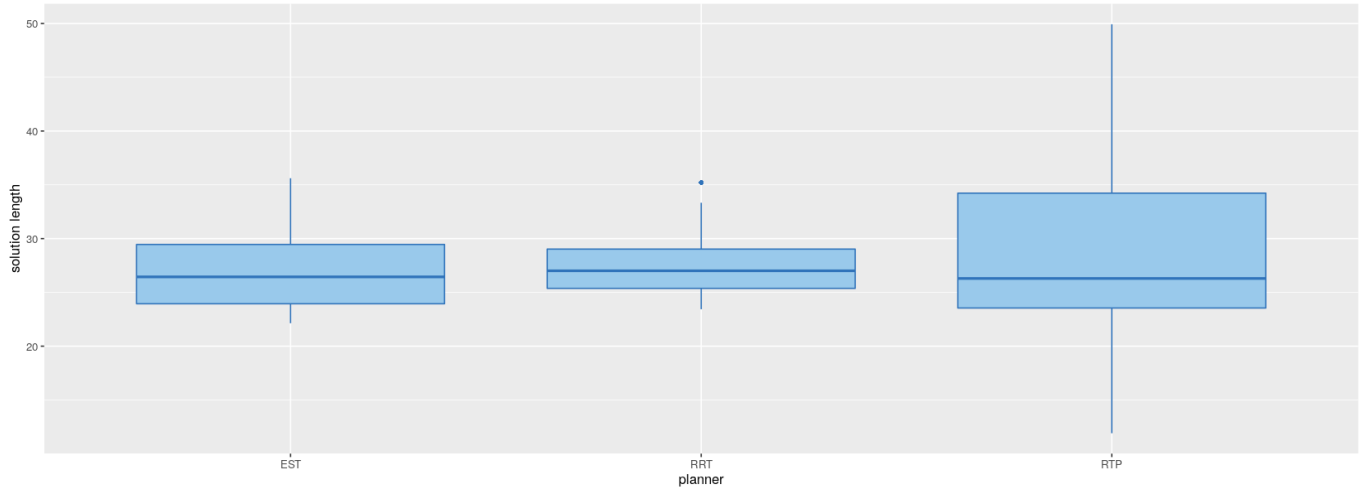


Benchmarking Plots for 20 Linked Chain

Time Benchmarking



Solution Length Benchmarking



Graph state Count Benchmarking