Introduction

In this notebook, we explore the problem of path planning within a defined environment using the Rapidly-exploring Random Tree (RRT) algorithm. The environment is a 10 x 10 square the goal and starting state predefined. These conditions can be changed freely.

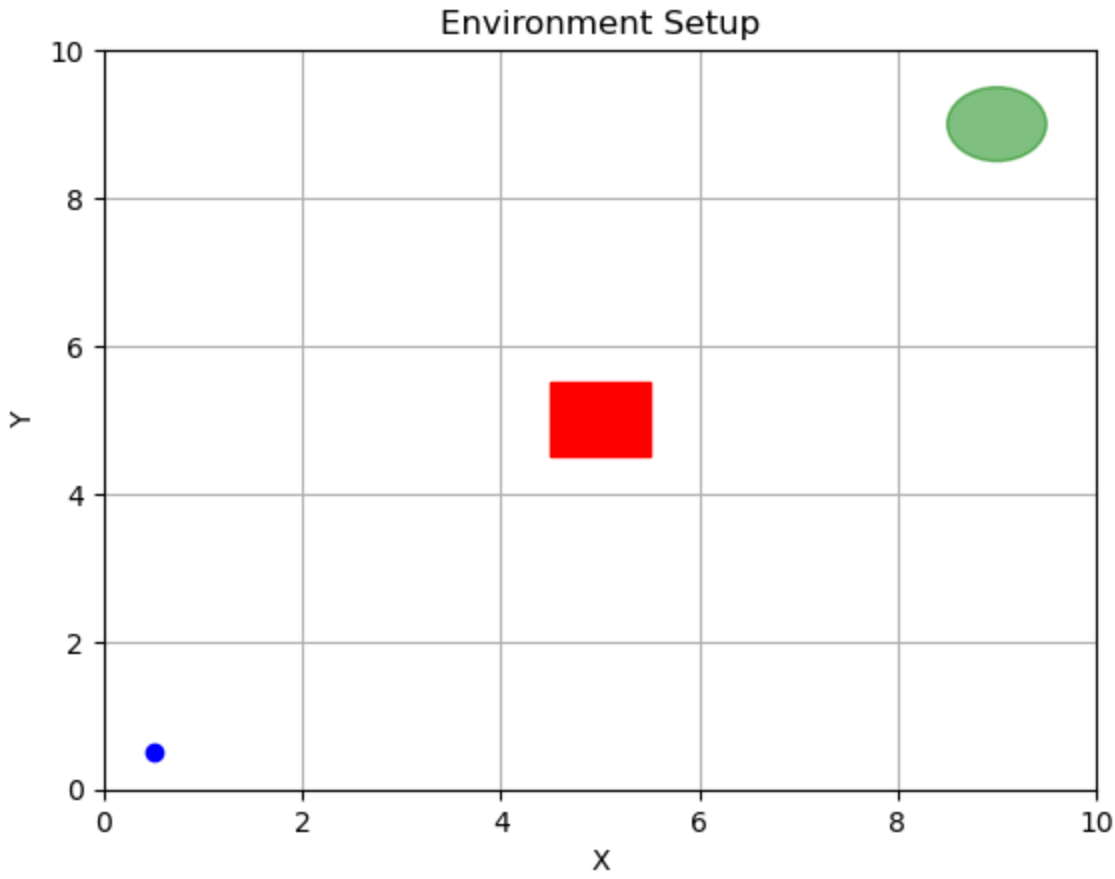
The goal is to experiment with the different parameters of the RRT algorithim to improve its preformance and effeciency.

The RRT algorithm is a popular method for path planning that rapidly explores the space by randomly generating points and connecting them in a tree-like structure. This notebook details the implementation of RRT and its application to the given problem, including parameter selection, algorithm execution, and performance evaluation.

By: Seif & Mark

In [9]:
```
# Importing necessary libraries
import matplotlib.pyplot as plt
from scipy.spatial import distance
import numpy as np
import random
import time
import pandas as pd
import seaborn as sns
```

In [10]:
```
# Environment parameters
x_min, x_max, y_min, y_max = 0, 10, 0, 10  # Environment dimensions
initial_state = (0.5, 0.5)  # Starting point
goal_center = (9, 9)  # Goal region center
goal_radius = 0.5  # Goal region radius
obstacle = plt.Rectangle((4.5, 4.5), 1, 1, color='red')  # Obstacle definition
goal = plt.Circle(goal_center, goal_radius, color='green', alpha=0.5) # goal d

# Plotting the environment
fig, ax = plt.subplots()
ax.add_patch(obstacle)
ax.add_patch(goal)
ax.plot(initial_state[0], initial_state[1], 'bo')  # Initial state
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.title('Environment Setup')
plt.xlabel('X')
plt.ylabel('Y')
plt.grid(True)
plt.show()
```



RRT Algorithm Implementation, Function Decleration

In [11]:
```
# Node class definition
class Node:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.parent = None

# Function for random sampling /// Using Goal Biasing
def random_sample():
    if random.random() > 0.1:  # 90% chance to sample randomly within the envi
        return random.uniform(x_min, x_max), random.uniform(y_min, y_max)
    else:  # 10% chance to sample the goal region
        return goal_center

# Function to find the nearest node
def nearest_node(nodes, sample):
    # Returning the nearest node to the generated random point
    return min(nodes, key=lambda node: distance.euclidean((node.x, node.y), sa

# Function to steer from the nearest node towards the sample
def steer(x_nearest, y_nearest, x_sample, y_sample, delta_distance=0.2):
    # Pointing to the next point
    theta = np.arctan2(y_sample - y_nearest, x_sample - x_nearest)
    # Returning a point one step size in the direction of random sampled point
    x_new = x_nearest + delta_distance * np.cos(theta)
```

```
    y_new = y_nearest + delta_distance * np.sin(theta)
    return x_new, y_new

def is_inside_any_obstacle(x, y, obstacles):
    for obstacle in obstacles:
        # Unpack the obstacle
        (ox, oy), width, height = obstacle

        # Check if the point is within the bounds of the obstacle
        if ox <= x <= ox + width and oy <= y <= oy + height:
            return True  # The point is inside this obstacle

    return False  # The point is not inside any obstacle

# Function to check if a point is inside the goal region
def is_inside_goal(x, y):
    return distance.euclidean((x, y), goal_center) <= goal_radius

# Function to check if a path between two nodes is valid (collision-free) consi
def is_path_valid(x1, y1, x2, y2, obstacles):
    points = np.linspace(0, 1, num=10)
    for point in points:
        x = x1 + (x2 - x1) * point
        y = y1 + (y2 - y1) * point
        if is_inside_any_obstacle(x, y, obstacles):
            return False
    return True

# The main RRT Function
def rrt(max_iterations=1000, obstacles=[], delta_distance=0.2):
    nodes = [Node(initial_state[0], initial_state[1])]
    for i in range(max_iterations):
        x_rand, y_rand = random_sample()
        nearest = nearest_node(nodes, (x_rand, y_rand))
        x_new, y_new = steer(nearest.x, nearest.y, x_rand, y_rand,delta_distan

        if not is_inside_any_obstacle(x_new, y_new, obstacles) and is_path_vali
            new_node = Node(x_new, y_new)
            new_node.parent = nearest
            nodes.append(new_node)

            if is_inside_goal(x_new, y_new):
                return nodes, new_node  # Path found

    return nodes, None  # Path not found within max iterations

def create_random_obstacles(num_obstacles, obstacle_size=1):
    obstacles = []
    while len(obstacles) < num_obstacles:
        obstacle_x = random.uniform(x_min + obstacle_size/2, x_max - obstacle_s
        obstacle_y = random.uniform(y_min + obstacle_size/2, y_max - obstacle_s

        # Check if the obstacle overlaps with the initial state or goal region
        if not (initial_state[0] - obstacle_size/2 <= obstacle_x <= initial_sta
                initial_state[1] - obstacle_size/2 <= obstacle_y <= initial_sta
           not (goal_center[0] - goal_radius - obstacle_size/2 <= obstacle_x <=
                goal_center[1] - goal_radius - obstacle_size/2 <= obstacle_y <=
            obstacles.append(((obstacle_x - obstacle_size/2, obstacle_y - obsta
    return obstacles
```

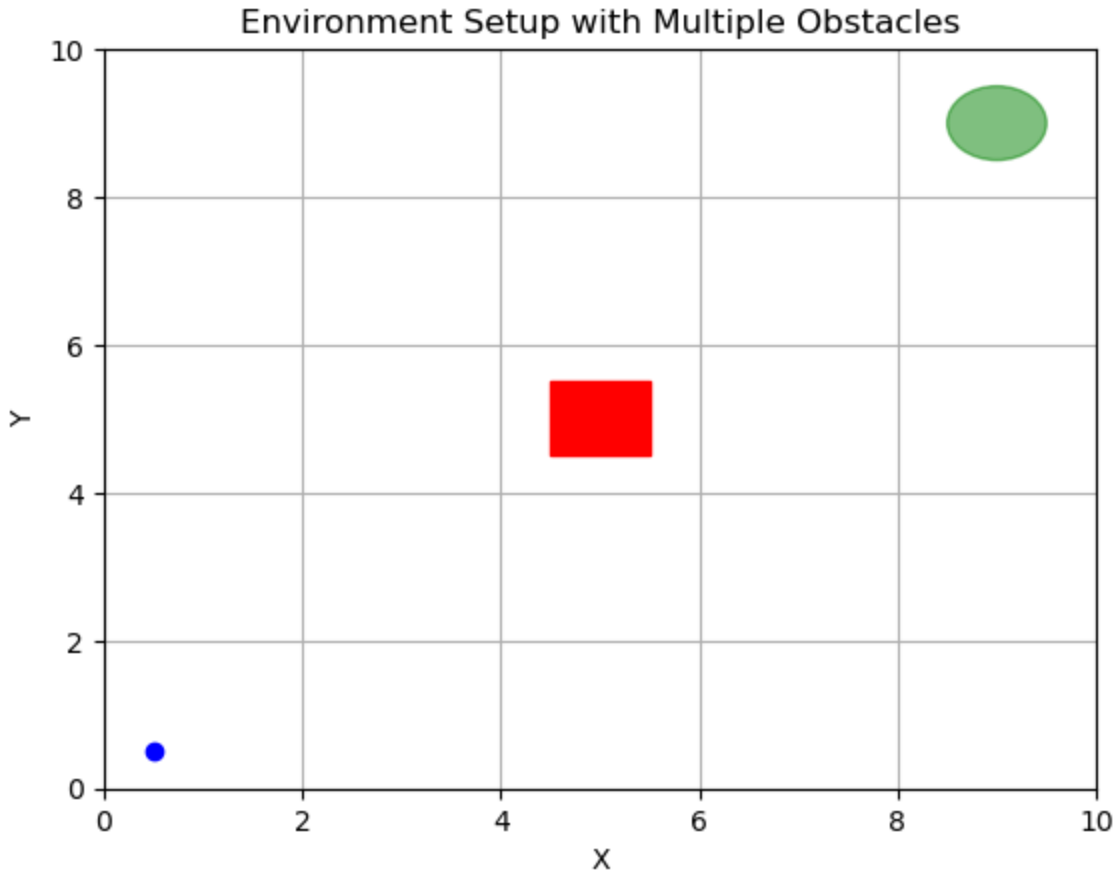
```
def plot_environment_with_obstacles(obstacles):
    fig, ax = plt.subplots()
    for obstacle in obstacles:
        ax.add_patch(plt.Rectangle(obstacle[0], obstacle[1], obstacle[2], colo
    ax.add_patch(plt.Circle(goal_center, goal_radius, color='green', alpha=0.5
    ax.plot(initial_state[0], initial_state[1], 'bo')  # Initial state
    plt.xlim(x_min, x_max)
    plt.ylim(y_min, y_max)
    plt.title('Environment Setup with Multiple Obstacles')
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.grid(True)
    plt.show()

def plot_environment(nodes, final_node=None, obstacles=[]):
    fig, ax = plt.subplots()

    # Plot all obstacles
    for obstacle in obstacles:
        ax.add_patch(plt.Rectangle(obstacle[0], obstacle[1], obstacle[2], colo

    # Plot the goal region
    ax.add_patch(plt.Circle(goal_center, goal_radius, color='green', alpha=0.5

    # Plot the initial state
    ax.plot(initial_state[0], initial_state[1], 'bo')

    # Plot all nodes and paths
    for node in nodes:
        if node.parent:
            plt.plot([node.x, node.parent.x], [node.y, node.parent.y], 'r-')

    # Highlight path to goal if found
    if final_node:
        path_node = final_node
        while path_node.parent:
            plt.plot([path_node.x, path_node.parent.x], [path_node.y, path_nod
            path_node = path_node.parent

    plt.xlim(x_min, x_max)
    plt.ylim(y_min, y_max)
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.title('RRT Path Planning')
    plt.grid(True)
    plt.show()
```

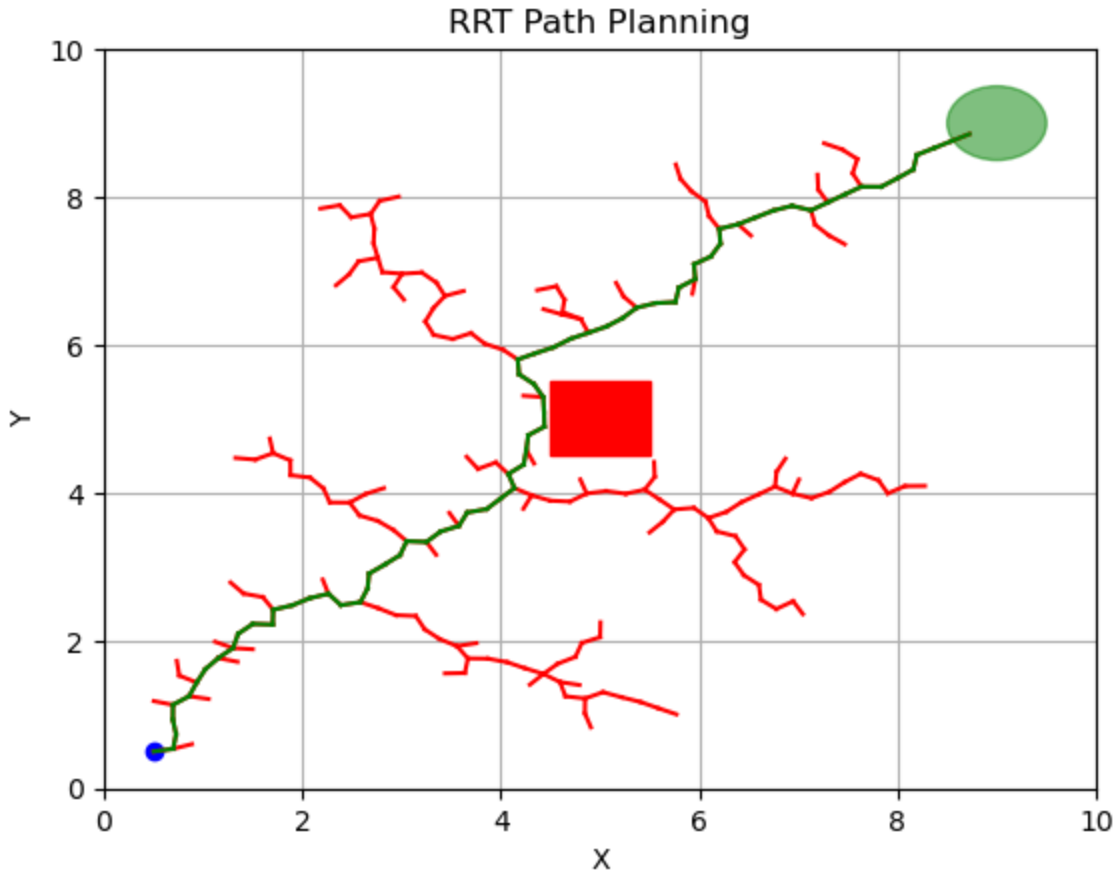
Running RRT for one obstacle

In [12]:

```
# Placing the obstacle manualy
obstacles = [((4.5, 4.5), 1, 1)]  # (x, y, width, height)

# Plotting the environment with just the obstacles
plot_environment_with_obstacles(obstacles)

# Runing the RRT algortihim
```

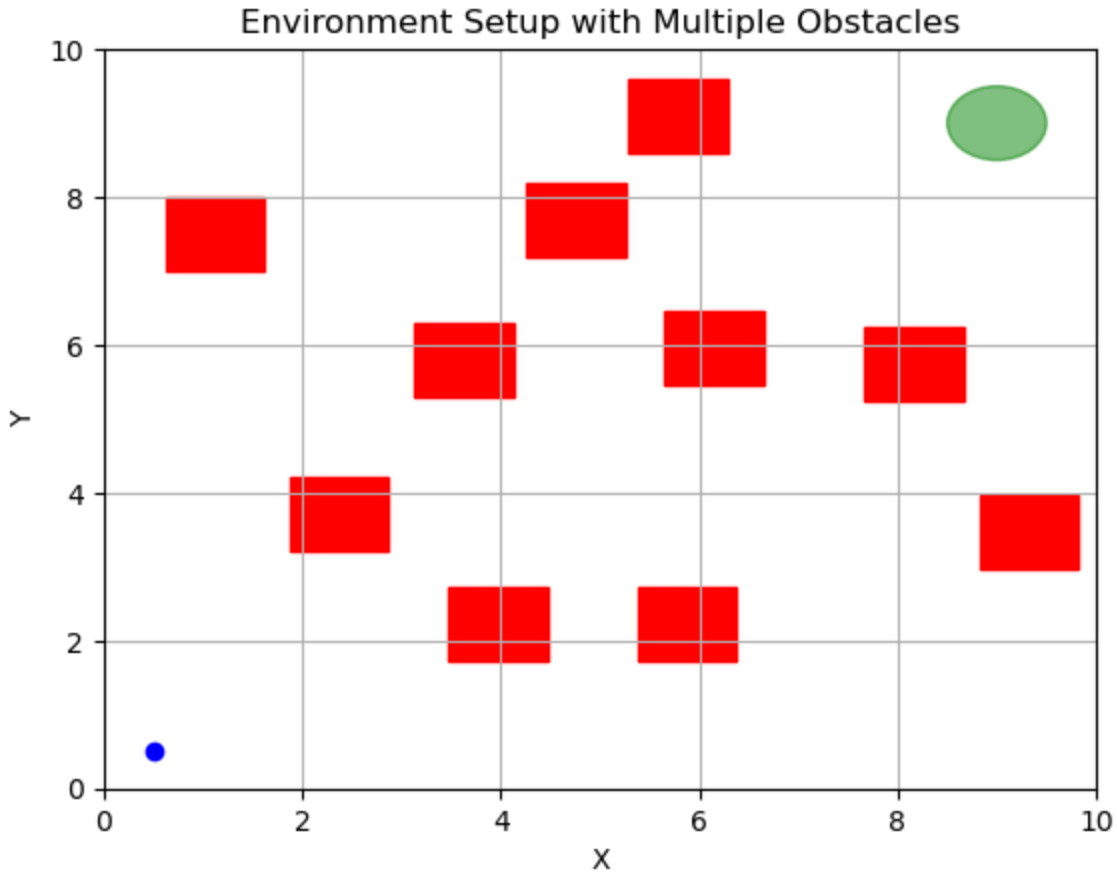
```
nodes, final_node = rrt(obstacles=obstacles)
plot_environment(nodes, final_node, obstacles)
```

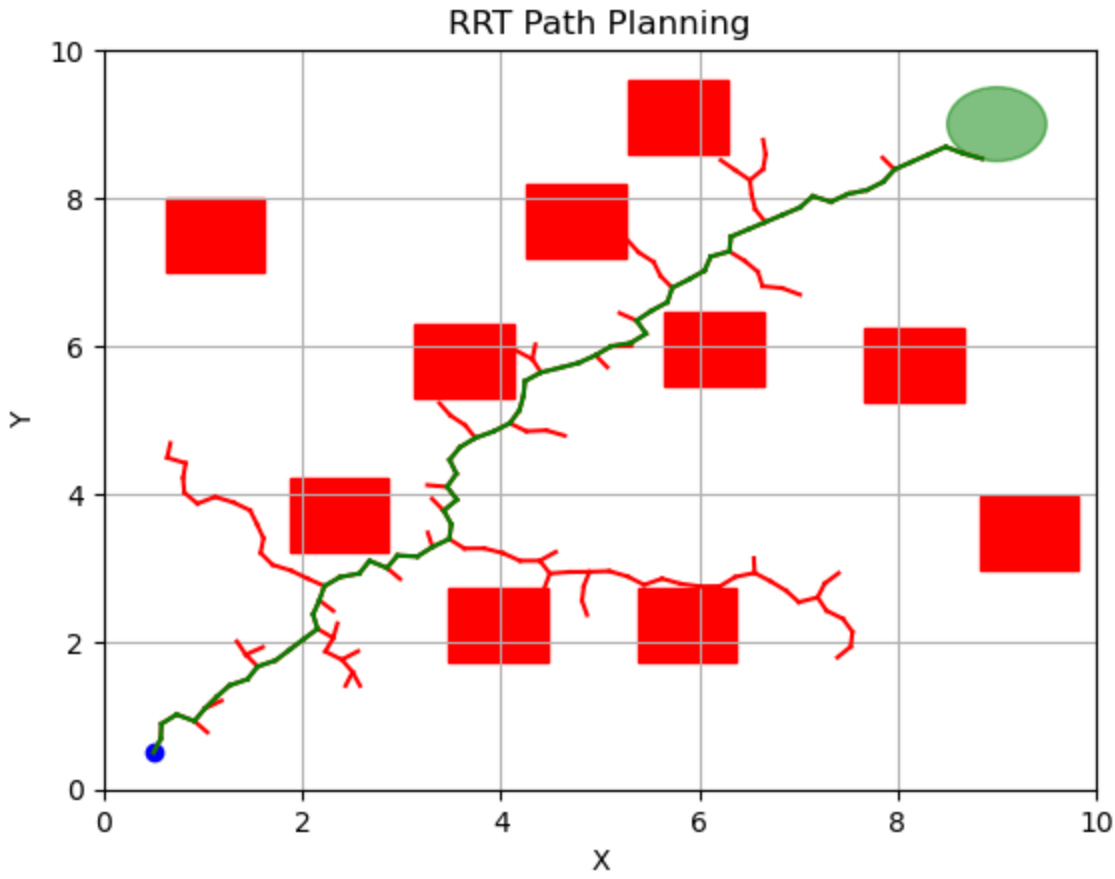




Running RRT for multiple obstacles

In [13]:
```
# Choosing the number of obstacles
num_obstacles = 10
obstacles = create_random_obstacles(num_obstacles)

# Plotting the environment with just the obstacles
plot_environment_with_obstacles(obstacles)

# Runing the RRT algortihim
nodes, final_node = rrt(obstacles=obstacles)
plot_environment(nodes, final_node, obstacles)
```

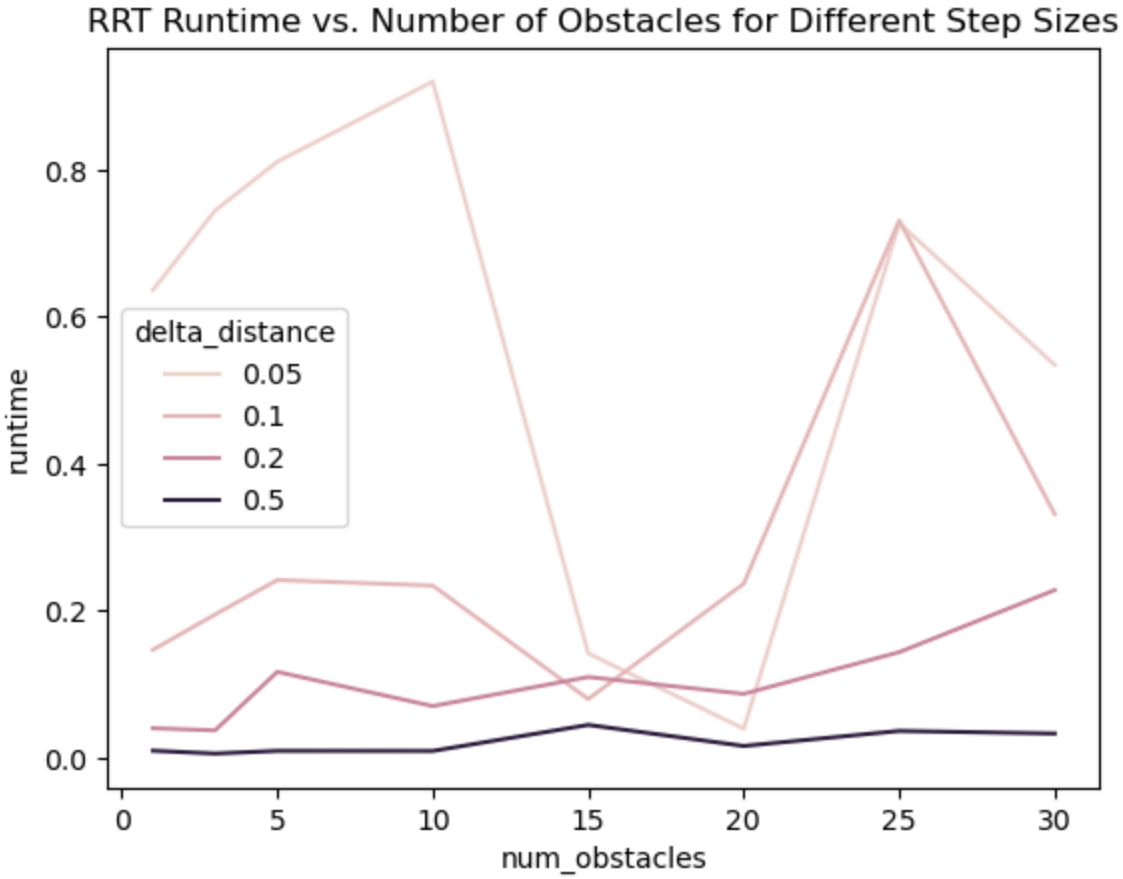




Analyzing preformance with different parameters

In [14]:
```
def run_experiments(num_obstacle_list, delta_distance_list, max_iterations=100(
    results = []

    for num_obstacles in num_obstacle_list:
        obstacles = create_random_obstacles(num_obstacles)

        for delta_distance in delta_distance_list:
            start_time = time.time()
            nodes, final_node = rrt(max_iterations=max_iterations, obstacles=ol
            end_time = time.time()

            runtime = end_time - start_time
            path_found = final_node is not None

            results.append({
                'num_obstacles': num_obstacles,
                'delta_distance': delta_distance,
                'runtime': runtime,
                'path_found': path_found
            })

    return results

# Defining the parameters to be tested
num_obstacle_list = [1, 3, 5, 10, 15, 20, 25, 30]
```

```
delta_distance_list = [0.05, 0.1, 0.2, 0.5]

# Running the experiment function
experiment_results = run_experiments(num_obstacle_list, delta_distance_list)

# Convert results to a DataFrame
df_results = pd.DataFrame(experiment_results)

# Plottig the results
sns.lineplot(data=df_results, x='num_obstacles', y='runtime', hue='delta_distar
plt.title('RRT Runtime vs. Number of Obstacles for Different Step Sizes')
plt.show()
```



Conclusion

In this notebook, we applied the RRT algorithm to solve a path planning problem in a specified environment. The algorithm successfully found paths from the initial state to the goal region, avoiding the both hard coded obstacles and randomly generated obstacles.

Goal Biasing was implemented to help steer the path towards the goal as a way to make the algorithim converge faster and use less resources.

Overall, the biggest factor in determining runtime was the step size between each sampling, as the bigger the step size the more the algorithim must sample which involves significantly more calculations and resources.