ÉCOLE NATIONALE SUPÉRIEURE DE TECHNIQUES AVANCÉES

ROB316 Planification et contrôle

TP - RRT

Luiz Henrique MARQUES GONÇALVES

Palaiseau, France 2024

1. RRT vs RRT*

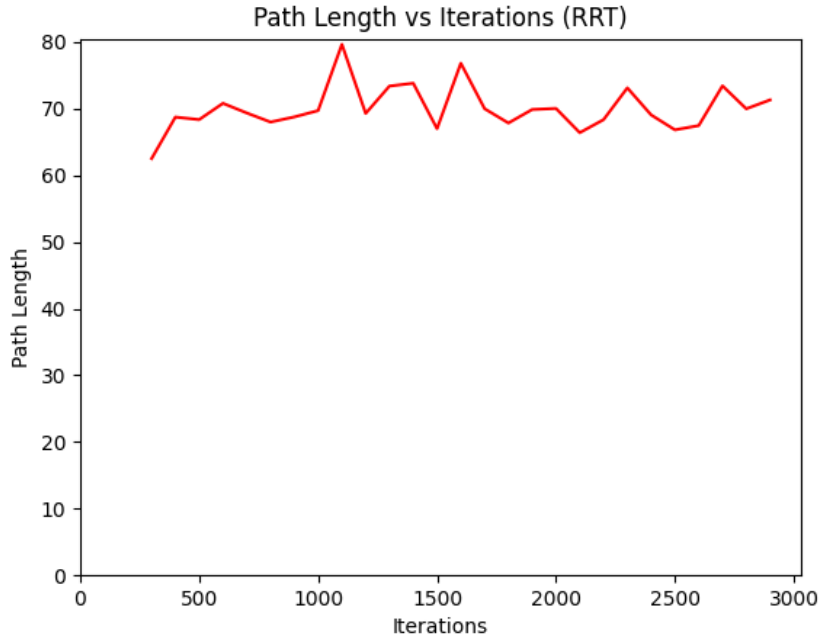
Question 1 : Test the two algorithms RRT and RRT* on this problem by varying the maximum number of iterations. What can you see on the average lengths of the paths ? On the computation times ? Remember to disable the display and to make several experiments to have significant results as these algorithms are stochastic

The main function was adapted to iterate with different maximum numbers of iterations (parameter *iter_max*) for the RRT function. At each *iter_max*, the operation was executed *times_per_max_iteration* times, and the average path length and average computation time were recorded. At the end of all iterations, the resulting graph was displayed. See the new code below:

```
def main():

    n_iters = 3000
    step_iter = 100
    lenghts = []
    times = []

    times_per_max_iteration = 5

    for iter in range(0, n_iters, step_iter):
        avg_length = 0
        n_success = 0
        avg_time = 0
        for i in range(times_per_max_iteration):
            print("Iteration: ", iter, " - ", i)

            x_start=(2, 2)  # Starting node
            x_goal=(49, 24)  # Goal node
            environment = env.Env()

            rrt = Rrt(environment, x_start, x_goal, 2, 0.10, iter)

            start_time = time.time()
            path, nb_iter = rrt.planning()
            end_time = time.time()

            avg_time += (end_time - start_time)/times_per_max_iteration

            if path:
                avg_length += get_path_length(path)
                n_success += 1

        if(n_success): lenghts.append(avg_length/n_success)
        else: lenghts.append(None)

        times.append(avg_time)

    plotting.plt.plot(range(0, n_iters, step_iter), lenghts, color='red')
    plotting.plt.xlabel('Iterations')
    plotting.plt.ylabel('Path Length')
    plotting.plt.title('Path Length vs Iterations (RRT)')
    plotting.plt.xlim(xmin=0)

    plotting.plt.figure()
    plotting.plt.plot(range(0, n_iters, step_iter), times, color='blue')
    plotting.plt.xlabel('Iterations')
```

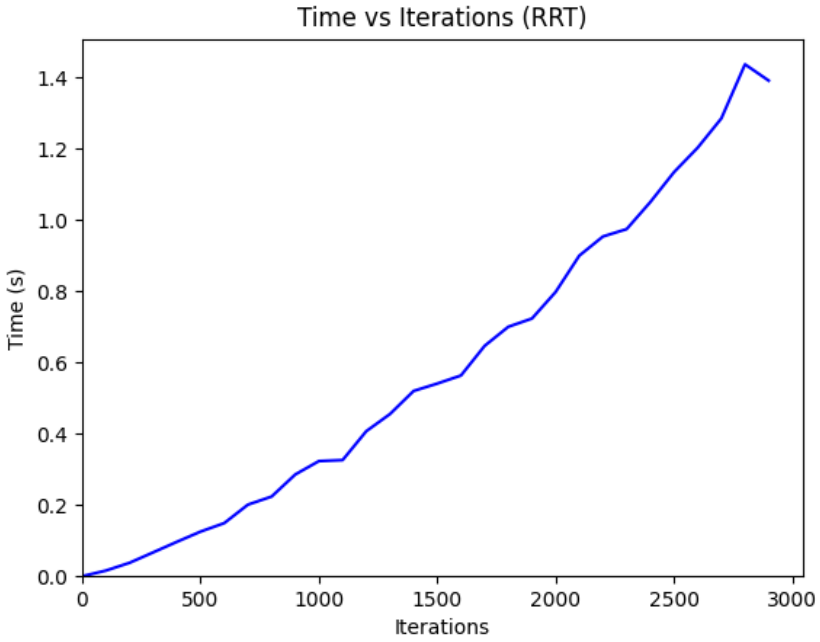
```
    plotting.plt.ylabel('Time (s)')
    plotting.plt.title('Time vs Iterations (RRT)')
    plotting.plt.xlim(xmin=0)

    plotting.plt.show()
```

For the RRT algorithm, the following graphs of average lengths and average computation times were obtained for a range between 100 and 3000 maximum iterations, with a step of 100 and *times_per_max_iteration = 10*:



As we can see, there is no significant relationship between path length and the number of iterations. We notice that there are no results for 100 and 200 iterations, which means no path was found. However, after a certain minimum number of iterations needed (in this case, 300), the path length remains constant.

These results were expected and can be explained by the fact that plain RRT does not take advantage of additional iterations to refine the path. Instead, it focuses on improving the map tree coverage. The first viable path



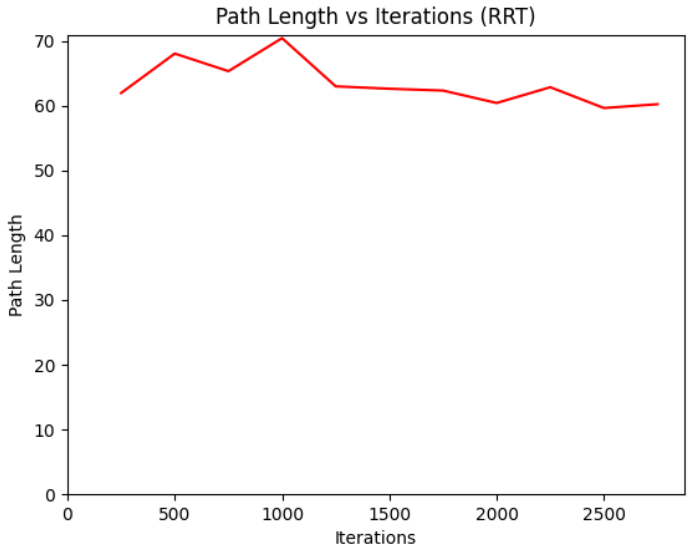
Unlike path length, computation time increases with the augmentation of the maximum number of iterations. Regardless of whether a path is found or not, a higher number of operations results in the creation of more nodes and a denser tree, which progressively requires more computational time.

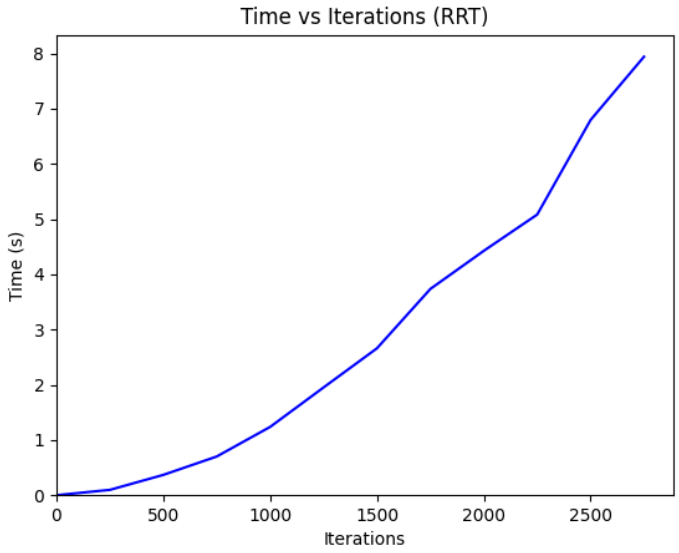
For the RRT* algorithm, the same process of iterations was applied. The structure is the same as the code shown above, with the only difference being the change from *rrt* to *rrt_star*.

Additionally, the following lines from the original *planning* function were commented out because they were computationally expensive and unnecessary for finding the optimal path (they are only required for the animation of intermediate paths).

```
            # index = self.search_goal_parent()
            # if index != (len(self.vertex) - 1) and iter_goal == None:
            #     iter_goal = k
```

The following graphs of average lengths and average computation times were obtained for a range between 250 and 3000 maximum iterations, with a step of 250 and *times_per_max_iteration* = *3*:





We can perceive a slight reduction in path length when the number of maximum iterations is raised. This was expected since, in this improved algorithm, at each new iteration, the path is locally updated to converge to lower-cost paths. As in RRT, time increases with the number of iterations.

Question 2 : Change the `step_len` parameter (default value is 2 in the provided code). What are the consequences of small values and large values on the two algorithms ?

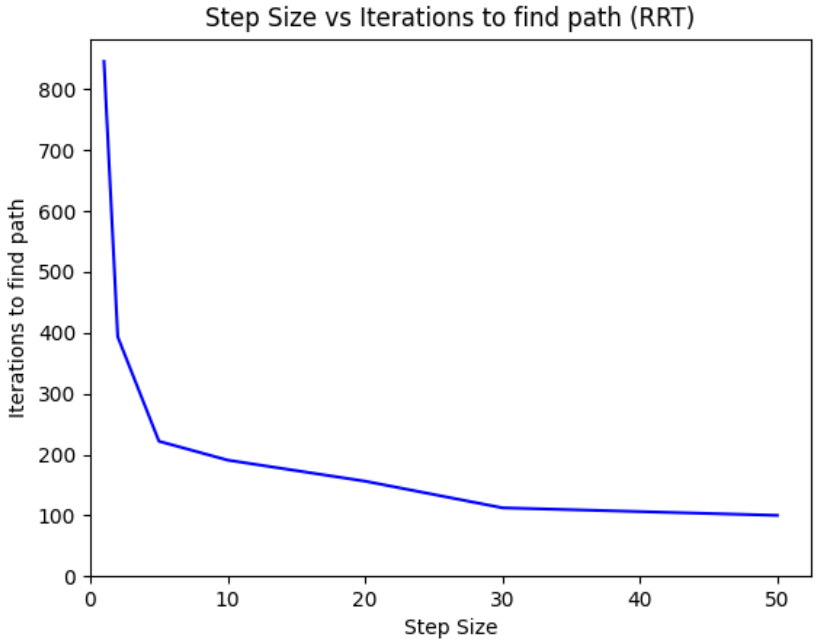
In the RRT algorithm, the main difference observed when changing step sizes was the number of iterations needed to find the path. Larger steps led to faster coverage of the map by the tree through big ”branches”, getting closer to the goal faster.

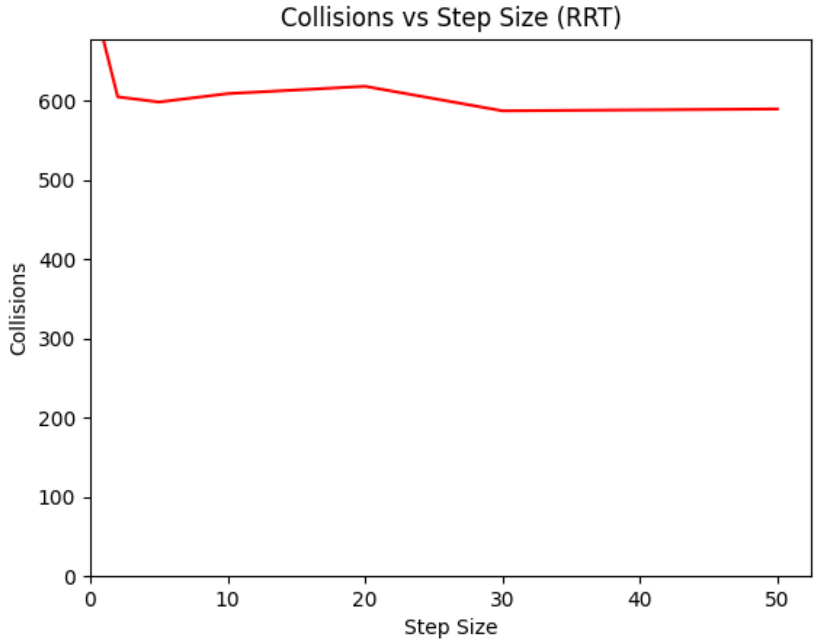
Intuitively, it was expected that the ”trade-off” is a higher number of collisions with obstacles when trying to extend the tree to new (and farther) nodes.

The following code was made to iterate over different step sizes and analyze the number of iterations needed to find the path and the number of collisions that occurred.

```
def main():

    list_collisions = []
    needed_iters = []

    steps = [1, 2, 5, 10, 20, 30, 50]

    times_per_stepsize = 10

    for step in steps:
        avg_collisions = 0
        avg_needed_iters = 0

        for i in range(times_per_stepsize):
            print("Iteration: ", step, " - ", i)

            x_start=(2, 2)  # Starting node
            x_goal=(49, 24)  # Goal node
            environment = env.Env()

            rrt = Rrt(environment, x_start, x_goal, step, 0.10, 2000)

            path, nb_iter, collisions = rrt.planning()

            if path:
                avg_collisions += collisions/times_per_stepsize
                avg_needed_iters += nb_iter/times_per_stepsize
                if showAnimation:
                    rrt.plotting.animation(rrt.vertex, path, "RRT", True)
                    plotting.plt.show()
        list_collisions.append(avg_collisions)
        needed_iters.append(avg_needed_iters)

    %Then collision and iterations plots.

```

The following results were obtained:





Indeed, larger steps led to faster coverage and path finding.

However, the hypothesis of a trade-off in collisions wasn't verified. No clear explanations were found or other "trade-offs" were identified within the scope of this work.

Similar results were obtained for RRT*.

2. Planification in narrow corridors

In this part, use the default starting and goal position provided in the code, the Env2 environment (`environment = env.Env2()`), and the RRT algorithm with the initial parameters (`rrt = Rrt(environment, x_start, x_goal, 2, 0.10, 1500)`). You will see that the narrow corridor in the middle of the map makes it difficult to find a path.

Question 3 : Explain why it is difficult to grow the tree rapidly in this environment (in particular think about what happens when the tree tries to grow towards a random point from the nearest node).

The RRT algorithm generates random points in the search space and attempts to connect the nearest tree node to the generated point. In an environment with a narrow corridor, most random points will fall outside the corridor. This means that tree growth attempts frequently result in collisions with obstacles or invalid attempts.

The valid region (the corridor) represents only a small fraction of the total space. This significantly reduces the probability of generating random points within the corridor that would allow efficient tree growth.

Furthermore, until the tree reaches the entrance of the corridor, it cannot explore the regions inside it. This further delays progress toward the goal.

Thus, an approach to address this issue would be to favor the selection of random points within these narrow regions, as the likelihood of this happening randomly is low. This can be achieved by periodically selecting points near obstacles, as will be done in a simplified manner in the following question.

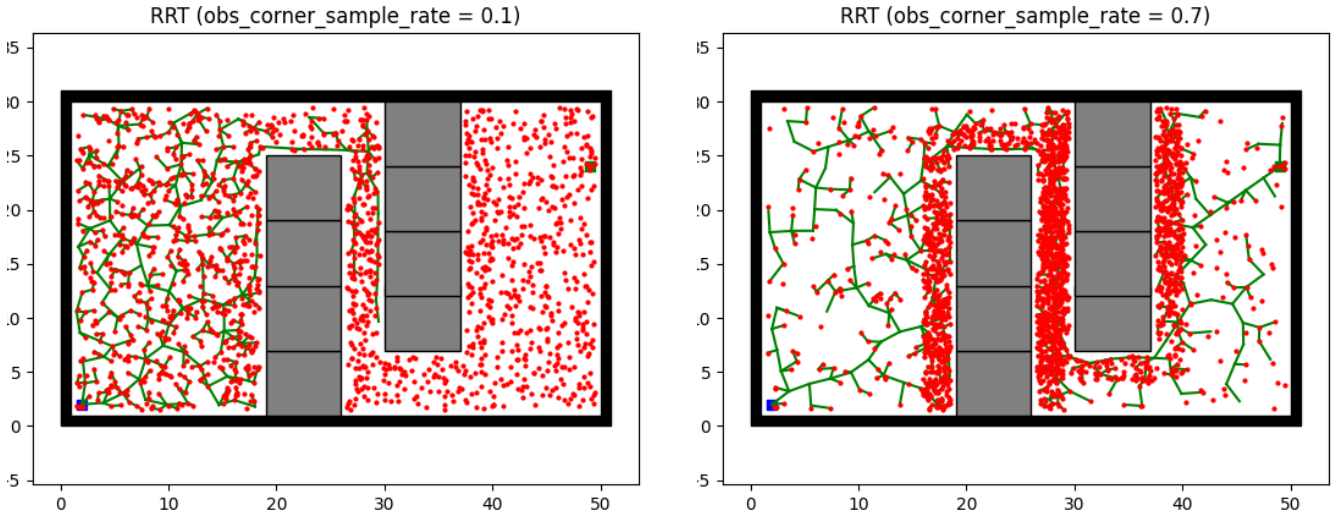
Question 4 : To improve this, modify the rrt.py file to implement a simple variant of the OBRRT algorithm. In this algorithm, the idea is to sample points taking into account the obstacles in order to increase the chances that the tree passes through difficult areas. Show the performance variation as a function of the percentage of points sampled using this strategy (from 0% to 100%).

The function *generate_random_node* was adapted to include a certain probability (*obs_corner_sample_rate*) of generating points near obstacles. This was achieved with the help of the function *generate_random_node_from_obs_corner*, which maps the available range and selects points near obstacles. It was implemented to consider all rectangle edges, not just the corners.

```python
def generate_random_node(self, obs_corner_sample_rate):

    rd_nb = np.random.random()

    if rd_nb < obs_corner_sample_rate:
        random_node = self.generate_random_node_from_obs_corner(self.max_corner_dist)

    elif rd_nb < obs_corner_sample_rate + self.goal_sample_rate:
        random_node = self.s_goal

    else:
        delta = self.utils.delta
        random_node = Node(
        (np.random.uniform(self.x_range[0] + delta, self.x_range[1] - delta),
         np.random.uniform(self.y_range[0] + delta, self.y_range[1] - delta)))

    if(not self.utils.is_inside_obs(random_node)):
        self.random_nodes.append(random_node)

    return random_node

```

```python
def generate_random_node_from_obs_corner(self, max_dist):

    delta = self.utils.delta
    x = None
    y = None

    # Select a random obstacle
    obs = self.env.obs_rectangle[np.random.randint(0, len(self.env.obs_rectangle))]

    # Define the range for x and y coordinates
    x_min = np.max((obs[0] - max_dist, self.x_range[0] + delta))
    x_max = np.min((obs[0] + obs[2] + max_dist, self.x_range[1] - delta))
    y_min = np.max((obs[1] - max_dist, self.y_range[0] + delta))
    y_max = np.min((obs[1] + obs[3] + max_dist, self.y_range[1] - delta))

    # Generate random coordinates within the defined range
    while x is None or self.utils.is_inside_obs(Node((x, y))):
        x = np.random.uniform(x_min, x_max)

        if x < obs[0] or x > obs[0] + obs[2]:
            y = np.random.uniform(y_min, y_max)
        else:
            y_top = np.random.uniform(obs[1] + obs[3], y_max)
            y_bottom = np.random.uniform(y_min, obs[1])
            y = np.random.choice([y_top, y_bottom])

    return Node((x, y))

```

To illustrate its functioning, the generated random points were plotted on the map for *obs_corner_sample_rate* values of 0.1 and 0.7, respectively:



((a)) *obs_corner_sample_rate* = 0.1 ((b)) *obs_corner_sample_rate* = 0.7

Figure 1: Comparison of generated random points for different *obs_corner_sample_rate* values.

As we can see, a higher sample rate near obstacles leads to denser trees in these areas.

To compare performance, the following code iterates over different sample rates from 0% to 100% with 10% steps and plots the success rates (which indicate the percentage of times, out of *times_for_rate* = *20*, that the algorithm found the path), keeping the proposed 1500 iterations:

```python
def main():

    corner_rate_step = 0.10
    times_for_rate = 20

    success = [0 for i in np.arange(0.0, 1.00 + corner_rate_step, corner_rate_step)]

    for obs_corner_sample_rate in np.arange(0.0, 1.0, corner_rate_step):
        print('Rate : ' + str(obs_corner_sample_rate))
        for i in range(times_for_rate):

            x_start=(2, 2)  # Starting node
            x_goal=(49, 24)  # Goal node
            environment = env.Env2()

            rrt = Rrt(environment, x_start, x_goal, 2, obs_corner_sample_rate, 1500)
            path, nb_iter = rrt.planning()

            if path:
                success[int(obs_corner_sample_rate/corner_rate_step)] += 1


    plotting.plt.plot(np.arange(0.0, 1.0 + corner_rate_step, corner_rate_step),
    np.array(success)/times_for_rate, color='red')
    plotting.plt.title("Success by new algorithm rate")
    plotting.plt.xlabel("Rate")
    plotting.plt.ylabel("Success")
    plotting.plt.show()

```

The following result was obtained:

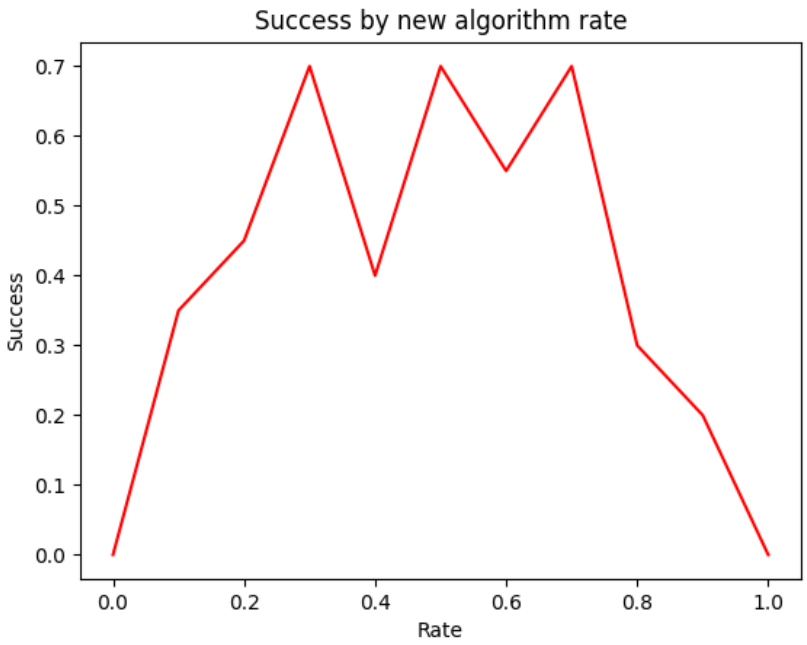


Figure 2: Success rate as a function of near-obstacle sample rates.

We can observe a success rate around 70% for near-obstacle sample rates in the 30%-70% range. In this range, the new solution appears to create enough points in narrow corridors without interrupting the tree's growth.

Small rates (0.0 - 0.2) demonstrate the previously discussed problem: a low probability of generating random points within the corridor inhibits efficient tree growth.

Additionally, this analysis reveals an interesting result for higher rates (0.7 - 1.0): although this allows good sampling in narrow areas, it becomes excessive and restricts the growth of the tree in these regions. This limitation prevents the tree from expanding into open areas towards the goal, reducing the success rate.