RBE 550 Motion Planning

Assignment 4
RRT and RRT *

Aadesh Varude
avarude@wpi.edu

RRT

- First the function gets a new point and gives a random point on the graph for the RRT search.

```
def get_new_point(self, goal_bias):
    new_x=np.random.randint(0,self.size_row-1)
    new_y=np.random.randint(0,self.size_col-1)
    new_node=Node(new_x,new_y)
    return new_node
```

- With the new found node we can calculate the nearest neighbor nodes, for which the function calculates and gathers nearest neighbor from the list of vertices.

```
def get_nearest_node(self, point):    min_distance=np.inf
    for i in self.vertices:
        dist=self.dis(point,i)
        if dist<min_distance:
            n_node=i
            min_distance=dist

    return n_node
```

- the get new node without collision selects a new node with some step size on the line joining the nearest node and the exploration node(the random point). It also checks for collision and checks for the connection to the goal.

```
def get_new_node_without_collision(self,node1,node2,stepsize=10):
    x1=node1.row
    x2=node2.row
    y1=node1.col
    y2=node2.col
    angle = m.atan2(y2-y1, x2-x1)
    new_x=x1+stepsize*np.cos(angle)
    new_y=y1+stepsize*np.sin(angle)
    if new_x>self.size_row:
        new_x=self.size_row-1
    if new_y>self.size_col:
        new_y=self.size_col-1
    new_node=Node(new_x,new_y)
    # check collision from node to the exploration goal
    # print(new_y,new_x)
    if(self.check_collision(new_node,node1)):

        node_connection=False
    else:
        node_connection=True

    # check collision from node to the goal
    if(self.check_collision(new_node,self.goal)):
        goal_connection=False
    else:
        goal_connection=True
    return new_node,node_connection,goal_connection
```

- **Main code :** In this portion, we first pick up a random node and then find the closed node to that node, and then we get the new node using the get nearest node function, later if we have the goal connection with less than 15 units we terminate our search as the goal is found, if node we extend the node and carry the search further.

```
     for i in range(n_pts):
        exp_node=self.get_new_point(0)
        n_node=self.get_nearest_node(exp_node)
        # print("exp_node",exp_node.row,exp_node.col)
        # print("n_node",n_node.row,n_node.col)
        new_node,node_connection,goal_connection=self.get_new_node_without_collision(n_node,exp_node)
        if(node_connection and goal_connection and self.dis(new_node,self.goal)<=15): # checking collision for the nearest neighbour and the new node
            # print("direct goal found")
            new_node.parent=n_node
            new_node.cost=new_node.parent.cost+self.dis(new_node,n_node)
```

```
                self.vertices.append(new_node)

                # if(self.found== False):
                self.goal.parent=new_node
                self.goal.cost=self.goal.parent.cost+self.dis(new_node,self.goal)
                self.vertices.append(self.goal)
                self.found=True
                break

            elif(node_connection):
                # print("node cosnnection found")
                new_node.parent=n_node
                new_node.cost=new_node.parent.cost+self.dis(new_node,n_node)
                self.vertices.append(new_node)
            else:
                continue
```

- Finally the path is backtracked using the parent nodes.

RRT *

- We follow the same steps as RRT till the new node is found.
- For the node connection extension we pick up the nearest neighbor node with the minimum cost node as the parent of the new node and then we rewire the tree if needed, we carry our search for n iterations in the loop.

```
        for i in range(n_pts):
            exp_node=self.get_new_point(0)
            n_node=self.get_nearest_node(exp_node)
            new_node,node_connection,goal_connection=self.get_new_node_without_collision(n_node,exp_node)
            if(node_connection and goal_connection and self.dis(new_node,self.goal)<=10): # checking collision for the nearest neighbour and the new node
                #Find the parent node with the least cost in the neighbor hood
                neighbor_nodes=self.get_neighbors(new_node,neighbor_size)
                min_parent_node=n_node
                new_node.parent=min_parent_node
                min_cost=new_node.parent.cost+self.dis(new_node,n_node)
                for i in neighbor_nodes:
                    if self.check_collision(i,new_node)==False:
                        distance=self.dis(i,new_node)
                        new_cost=i.cost+distance
                        if new_cost<min_cost:
                                min_parent_node=i
                                min_cost=new_cost
                new_node.parent=min_parent_node
                new_node.cost=min_cost
                self.vertices.append(new_node)
                self.rewire(new_node, neighbor_nodes)

                if(self.found== False):
                    neighbor_nodes=self.get_neighbors(self.goal,neighbor_size)
                    min_parent_node=new_node
                    self.goal.parent=min_parent_node
                    min_cost=self.goal.parent.cost+self.dis(self.goal,n_node)
                    for i in neighbor_nodes:
                        if self.check_collision(i,self.goal)==False:
                            distance=self.dis(i,self.goal)
                            new_cost=i.cost+distance
                            if new_cost<min_cost:
                                    min_parent_node=i
                                    min_cost=new_cost
                    self.goal.parent=min_parent_node
                    self.goal.cost=min_cost
                    self.vertices.append(self.goal)
                    self.found=True

            elif(node_connection):
                neighbor_nodes=self.get_neighbors(new_node,neighbor_size)
                #Find the parent node with the least cost in the neighbor hood
                min_parent_node=n_node
                new_node.parent=min_parent_node
                min_cost=new_node.parent.cost+self.dis(new_node,n_node)
                for i in neighbor_nodes:
                    if self.check_collision(i,new_node)==False:
                        distance=self.dis(i,new_node)
                        new_cost=i.cost+distance
                        if new_cost<min_cost:
                                min_parent_node=i
                                min_cost=new_cost
                new_node.parent=min_parent_node
                new_node.cost=min_cost
                self.vertices.append(new_node)
                self.rewire(new_node, neighbor_nodes)
            else:
                continue
```

- The major difference between RRT and RRT * is the rewire and get neighbor functions.
- The rewire function checks the neighbors for a lower cost than the current cost to rewire the tree with the parent as the new node.

```
def rewire(self, new_node, neighbors):
    for i in neighbors:
        distance=self.dis(i,new_node)
        new_cost=new_node.cost+distance
        if i.cost>new_cost:
            if self.check_collision(i,new_node)==False:
                # print("changing parent")
                i.cost=new_cost
                i.parent=new_node
```

- The get neighbor function returns the neighbors in the given radius (neighbor size).

```
def get_neighbors(self, new_node, neighbor_size):
    neighbor_nodes=[]
    for i in self.vertices:
        if self.dis(new_node,i)<neighbor_size:
            neighbor_nodes.append(i)
    return neighbor_nodes
```

Q1 For RRT, what is the main difference between RRT and RRT*? What change does it make in terms of the efficiency of the algorithms and optimality of the search result?

The main difference between RRT and RRT * is the rewiring step where the tree is rewired for having the minimum cost, another difference is in RRT we pick the nearest node but in RRT * we check all the near neighbors and pick up the node with the minimum cost as the parent of the new node. In terms of efficiency, RRT is generally faster than RRT* because it does not perform the optimization step. However, RRT* can find a more optimal solution than RRT.

Q2 Compare RRT with the results obtained with PRM in the previous assignment. What are the advantages and disadvantages?
For uniform sampling, we get the same path in almost all the runs as the algorithm searches the entire space uniformly and according to path length, it gives comparable results with RRT*. For the gaussian and bridge sampling one needs more samples whereas, in the RRT family, you can get it with a lesser number of samples. The PRM uniform sampling has path length as 256 , the random sampling has path length as 282, the gaussian sampling path length is 318, and the bridge sampling path length is 267. The path length using RRT is 311 and the path length using RRT* is 266. Here the lengths of the path of all the algorithms are mentioned if one observes the path length of uniform and bridge are comparable with the RRT* and otherwise RRT* performs better than gaussian and random. One crucial observation for the RRT* is the more number of samples you add you tend to get a more optimal solution, whereas in bridge and gaussian sampling that might not be the case.
The advantages of PRM can be defined as :

- It explores the entire state space whereas the RRT leaves some regions of the map unexplored for the same number of iterations.
- PRM generates an optimal solution.
- PRM works well for multi-query problems meaning the root needs to find a path for the multiple start and goal configurations.

The disadvantages of PRM are as follows:

- PRM takes a good amount of computation time for higher dimensions.
- PRM algorithm requires pre-processing.
- Less effective in the dynamic environment

The advantages of RRT can be defined as :

- It is efficient in high dimensional space with complex obstacles.
- It can generate a path even in dynamic obstacles environment.
- RRT explores the environment without any pre-processing.

The disadvantages of RRT are as follows:

- RRT gives you a suboptimal solution.
- RRT can leave some areas while exploration that leads to limited coverage.

The advantages of RRT* can be defined as :

- It does have all the advantages of RRT.
- RRT* is probabilistically complete, if given enough time for exploration.
- RRT* is able to find an optimal path, whereas RRT gives a sub-optimal path.
- RRT* is a single query algorithm whereas PRM can work better for multi-query.

The disadvantages of RRT* are as follows:

- While RRT* does not have a guarantee on the computational time (meaning you cannot figure out the runtime) to find a solution.
- RRT* depends greatly on the choice of parameters.

Results

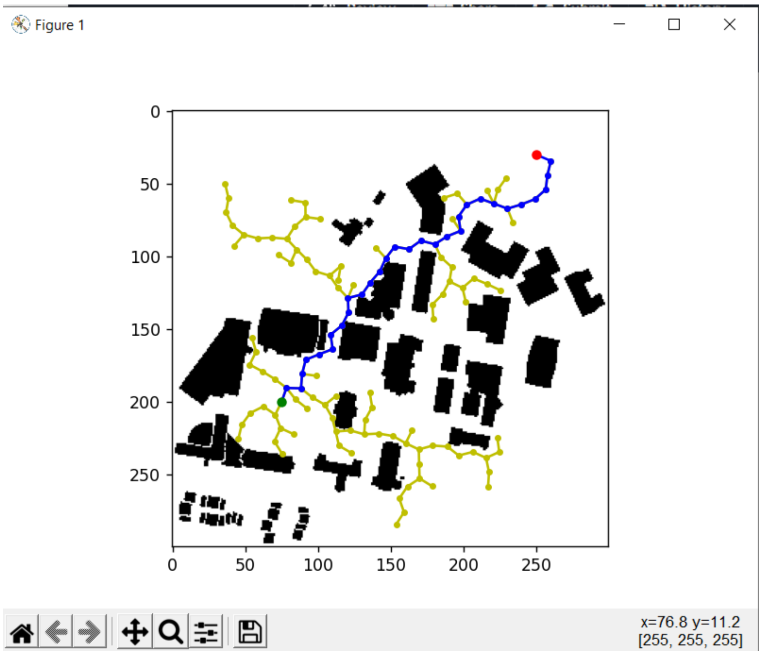

Figure 1: Image of the path after RRT.

```
It took 123 nodes to find the current path
The path length is 311.12
```

Figure 2: The number of nodes and path for the above path.

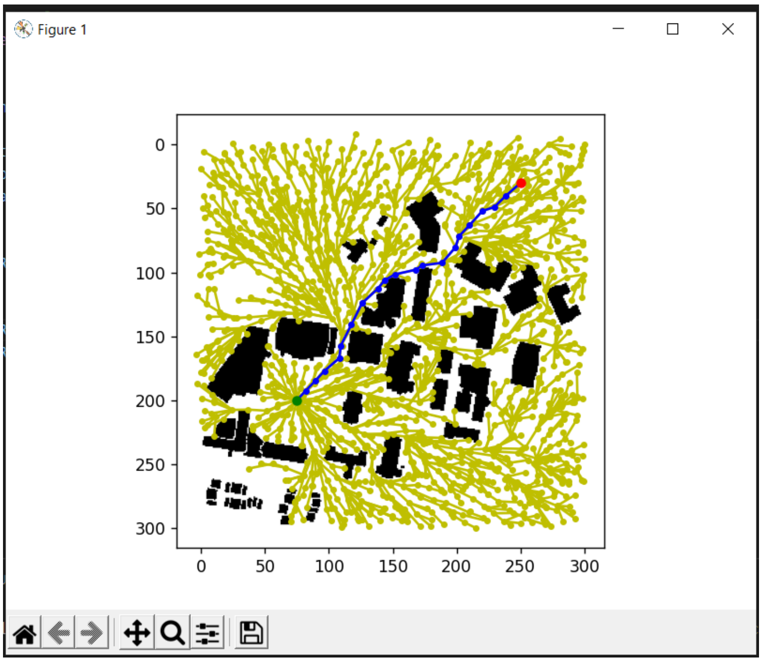

Figure 3: Image of the path after RRT *.

```
It took 1459 nodes to find the current path
The path length is 266.60
```

Figure 4: The number of nodes and path for the above path. This path is for the full number of times it should be sample i.e 2000.

References

1. https://gitee.com/HaiFengZhiJia/PythonRobotics/tree/master/PathPlanning
2. https://github.com/zhm-real/PathPlanning
3. https://github.com/aniketmpatil/standard-search-algorithms
4. Class lecture slides